Network Working Group C. Bormann
Internet-Draft Universität Bremen TZI
Intended status: Standards Track S. Leonard
Expires: 4 January 2021 Penango, Inc.
3 July 2020

Concise Binary Object Representation (CBOR) Tags for Object Identifiers
draft-bormann-cbor-tags-oid-07

Abstract

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

The present document defines CBOR tags for object identifiers (OIDs). It is intended as the reference document for the IANA registration of the CBOR tags so defined.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

1.1. Terminology

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

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

2. Object Identifiers

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

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

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

    X * 40 + Y

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

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

Tag TBD111: tags a byte string as the [X.690] encoding of an absolute object identifier (simply "object identifier" or "OID").

Tag TBD110: tags a byte string as the [X.690] encoding of a relative object identifier (also "relative OID"). Since the encoding of each number is the same as for [RFC6256] Self-Delimiting Numeric Values (SDNVs), this tag can also be used for tagging a byte string that contains a sequence of zero or more SDNVs.

2.1. Requirements on the byte string being tagged

A byte string tagged by TBD111 or TBD110 MUST be a syntactically valid BER representation of an object identifier: A concatenation of zero or more SDNV values, where each SDNV value is a sequence of one or more bytes that all have their most significant bit set, except for the last byte, where it must be unset; the first byte of each SDNV cannot be 0x80 (which would be a leading zero in SDNV's base-128 arithmetic).

In other words:

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

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

If either of these invalid conditions are encountered, the tag is invalid.

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

To enable the search for specific object ID values, it is RECOMMENDED that definite length encoding (see Section 2.2.2 of [RFC7049]) is used for the byte strings used as tag content for these tags.

The valid set of byte strings can also be expressed using regular expressions on bytes, using no specific notation but resembling [PCRE]. Unlike typical regular expressions that operate on character sequences, the following regular expressions take bytes as their domain, so they can be applied directly to CBOR byte strings.

For byte strings with tag TBD111:

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

For byte strings with tag TBD110:

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

A tag with tagged content that does not conform to the applicable regexp is invalid.

3. Examples

3.1. Encoding of the SHA-256 OID

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

Dotted Decimal Notation: 2.16.840.1.101.3.4.2.1

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

Figure 1: SHA-256 OID in BER

```
D8 6F                           # tag(111)
   49                           # 0b010_01001: mt 2, 9 bytes
      60 86 48 01 65 03 04 02 01  # X.690 Clause 8.19
```

Figure 2: SHA-256 OID in CBOR

3.2. Encoding of a MIB Relative OID

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

ASN.1 Value Notation: { lowpanObjects(1) lowpanStats(1) lowpanOutTransmits(29) }

Dotted Decimal Notation: .1.1.29

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

Figure 3: MIB relative object identifier, in BER

```
D8 6E                           # tag(110)
   43                           # 0b010_01001: mt 2 (bstr), 3 bytes
      01 01 1D                  # X.690 Clause 8.20
```

Figure 4: MIB relative object identifier, in CBOR

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

4. Discussion

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

5. Tag Factoring with OID Arrays and Maps

TBD111 and TBD110 can tag CBOR arrays and maps. The idea is that the tag is factored out from each individual byte string; the tag is placed in front of the array or map instead. The tags TBD111 and TBD110 are left-distributive.

When the TBD111 or TBD110 tag is applied to an array, it means that the respective tag is imputed to all items in the array that are byte strings. For example, when the array is tagged with TBD111, every array item that is a binary string is an OID.

When the TBD111 or TBD110 tag is applied to a map, it means that the respective tag is imputed to all keys in the map that are byte strings. The values in the map are not considered specially tagged.

Array and map nesting is permitted. For example, a 3-dimensional array of OIDs can be composed by using a single TBD111 tag, followed by an array of arrays of arrays of binary strings. All such binary strings are considered OIDs.
// That was part of the original proposal. I find it hard to imagine
// how to stop the influence of the tag deep into a nested structure.
// That's why I would rather limit this to one level (no nesting).
// But see the Figure below, which needs a nesting of two. Please
// discuss.

6. Applications and Examples of OIDs

6.1. X.500 Distinguished Name

Consider the X.500 distinguished name:

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

Table 1: Example X.500 Distinguished Name

Table 1 has four "relative distinguished names" (RDNs). The country and street RDNs are single-valued. The second and fourth RDNs are multi-valued.

The equivalent representations in CBOR diagnostic notation and CBOR are:

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

Figure 5: Distinguished Name, in CBOR Diagnostic Notation

```
d8 6f                                     # tag(111)
   84                                     # array(4)
      a1                                  # map(1)
         43 550406                        # 2.5.4.6 (4)
         62                               # text(2)
            5553                          # "US"
      a3                                  # map(3)
         43 550407                        # 2.5.4.7 (4)
         6b                               # text(11)
            4c6f7320416e67656c6573        # "Los Angeles"
         43 550408                        # 2.5.4.8 (4)
         62                               # text(2)
            4341                          # "CA"
         43 550411                        # 2.5.4.17 (4)
         65                               # text(5)
            3930303133                    # "90013"
      a1                                  # map(1)
         43 550409                        # 2.5.4.9 (4)
         6e                               # text(14)
            3533322053204f6c697665205374  # "532 S Olive St"
      a2                                  # map(2)
         43 55040f                        # 2.5.4.15 (4)
         6b                               # text(11)
            5075626c6963205061726b        # "Public Park"
         4a 0992268993f22c640130    # 0.9.2342.19200300.100.1.48 (11)
         6f                               # text(15)
            5065727368696e6720537175617265 # "Pershing Square"
```

Figure 6: Distinguished Name, in CBOR (109 bytes)

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

7. CDDL Control Operators

CDDL specifications may want to specify the use of SDNVs or SDNV sequences (as defined for the tag content for TBD110). This document introduces two new control operators that can be applied to a target value that is a byte string:

* ".sdnv", with a control type that contains unsigned integers. The byte string is specified to be encoded as an [RFC6256] SDNV (BER encoding) for the matching values of the control type.

* ".sdnvseq", with a control type that contains arrays of unsigned integers. The byte string is specified to be encoded as a sequence of [RFC6256] SDNVs (BER encoding) that decodes to an array of unsigned integers matching the control type.

Figure 7 shows an example for the use of ".sdnvseq" for a part of a structure using OIDs that could be used in Figure 6.
// We could define another control operator that includes the X*40+Y
// magic, so the example can actually use "[2, 5, 4, 6]". We could
// also add an operator that parses dotted decimal integer sequences,
// so we can use "2.5.4.6". I don’t see a strong reason for that.

```
country-rdn = {country-oid => country-value}
country-oid = bytes .sdnvseq [85, 4, 6]
country-value = text .size 2
```

Figure 7: Using .sdnvseq

8. IANA Considerations

8.1. CBOR Tags

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

Tag	Data Item	Semantics
TBD111	multiple	object identifier (BER encoding)
TBD110	multiple	relative object identifier (BER encoding); SDNV [RFC6256] sequence

Table 2: Values for New Tags

8.2. CDDL Control Operators

IANA is requested to assign the CDDL Control Operators in Table 3, with the present document as the specification reference.

Name	Reference
.sdnv	[this document, Section 7]
.sdnvseq	[this document, Section 7]

Table 3: New CDDL Operators

9. Security Considerations

The security considerations of RFC 7049 apply.

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

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

9.1. Conversions Between BER and Dotted Decimal Notation

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

10. References

10.1. Normative References

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

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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

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

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

10.2. Informative References

[DOUBLEDABBLE]
Gao, S., Al-Khalili, D., and N. Chabini, "An improved BCD adder using 6-LUT FPGAs", DOI 10.1109/newcas.2012.6328944, 10th IEEE International NEWCAS Conference, June 2012, <https://doi.org/10.1109/newcas.2012.6328944>.

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

[PCRE] Ho, A., "PCRE - Perl Compatible Regular Expressions", 2018, <http://www.pcre.org/>.

[PKILCAKE] Kaminsky, D., Patterson, M., and L. Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure", DOI 10.1007/978-3-642-14577-3_22, Financial Cryptography and Data Security pp. 289-303, 2010, <https://doi.org/10.1007/978-3-642-14577-3_22>.

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

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

Appendix A. Change Log

This section is to be removed before publishing as an RFC.

A.1. Changes from -06 to -07

Reduce the draft back to its basic mandate: Describe CBOR tags for what is colloquially know as ASN.1 Object IDs.

A.2. Changes from -05 to -06

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

A.3. Changes from -04 to -05

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

A.4. Changes from -03 to -04

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

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

* Rewrote the abstract to describe the content more accurately, and to point out that no changes to the wire protocol are being proposed.

* Removed "ASN.1" from "object identifiers", as OIDs are independent of ASN.1.

* Rewrote the introduction to be more about the present text.

* Proposed a concise OID arc.

* Provided binary regular expression forms for OID validation.

* Updated IANA registration tables.

A.5. Changes from -02 to -03

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

* Expanded the draft scope to be a comprehensive CBOR update.

* Added OID-related sections: OID Enumerations, OID Maps and Arrays, and Applications and Examples of OIDs.

* Added Tag 36 update (binary MIME, better definitions).

* Added stub/experimental sections for X.690 Series Tags (tag <<X>>) and Regular Expressions (tag 35).

* Added technique for representing sets and multisets.

* Added references and fixed typos.

Authors' Addresses

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

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

Sean Leonard
Penango, Inc.
5900 Wilshire Boulevard
21st Floor
Los Angeles, CA, 90036
United States of America

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

Network Working Group   C. Bormann
Internet-Draft   Universitaet Bremen TZI
Obsoletes: 7049 (if approved)   P. Hoffman
Intended status: Standards Track   ICANN
Expires: 19 December 2020   17 June 2020

Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-14

Abstract

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

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

Contributing

This note is to be removed before publishing as an RFC.

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.

Table of Contents

1. Introduction

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

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

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

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

1.1. Objectives

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

1. The representation must be able to unambiguously encode most common data formats used in Internet standards.

* It must represent a reasonable set of basic data types and structures using binary encoding. "Reasonable" here is largely influenced by the capabilities of JSON, with the major addition of binary byte strings. The structures supported are limited to arrays and trees; loops and lattice-style graphs are not supported.

* There is no requirement that all data formats be uniquely encoded; that is, it is acceptable that the number "7" might be encoded in multiple different ways.

2. The code for an encoder or decoder must be able to be compact in order to support systems with very limited memory, processor power, and instruction sets.

   * An encoder and a decoder need to be implementable in a very small amount of code (for example, in class 1 constrained nodes as defined in [RFC7228]).

   * The format should use contemporary machine representations of data (for example, not requiring binary-to-decimal conversion).

3. Data must be able to be decoded without a schema description.

   * Similar to JSON, encoded data should be self-describing so that a generic decoder can be written.

4. The serialization must be reasonably compact, but data compactness is secondary to code compactness for the encoder and decoder.

   * "Reasonable" here is bounded by JSON as an upper bound in size, and by the implementation complexity limiting how much effort can go into achieving that compactness. Using either general compression schemes or extensive bit-fiddling violates the complexity goals.

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

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

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

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

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

   * The format is designed for decades of use.

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

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

1.2. Terminology

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

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

This specification makes use of the following terminology:

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

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

Encoder: A process that generates the (well-formed) representation format of a CBOR data item from application information.

Data Stream: A sequence of zero or more data items, not further assembled into a larger containing data item (see [RFC8742] for one application). The independent data items that make up a data stream are sometimes also referred to as "top-level data items".

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

Valid: A data item that is well-formed and also follows the semantic restrictions that apply to CBOR data items (Section 5.3).

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

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

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

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

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

2. CBOR Data Models

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

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

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

* an integer in the range -2**64..2**64-1 inclusive

* a simple value, identified by a number between 0 and 255, but distinct from that number itself

* a floating-point value, distinct from an integer, out of the set representable by IEEE 754 binary64 (including non-finites) [IEEE754]

* a sequence of zero or more bytes ("byte string")

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

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

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

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

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

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

2.1. Extended Generic Data Models

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

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

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

* application data types such as a point in time or an RFC 3339 date/time string (tag numbers 1, 0)

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

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

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

2.2. Specific Data Models

The specific data model for a CBOR-based protocol usually subsets the extended generic data model and assigns application semantics to the data items within this subset and its components. When documenting such specific data models, where it is desired to specify the types of data items, it is preferred to identify the types by the names they have in the generic data model ("negative integer", "array") instead of by referring to aspects of their CBOR representation ("major type 1", "major type 4").

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

3. Specification of the CBOR Encoding

A CBOR data item (Section 2) is encoded to or decoded from a byte string carrying a well-formed encoded data item as described in this section. The encoding is summarized in Table 7, indexed by the initial byte. An encoder MUST produce only well-formed encoded data items. A decoder MUST NOT return a decoded data item when it encounters input that is not a well-formed encoded CBOR data item (this does not detract from the usefulness of diagnostic and recovery tools that might make available some information from a damaged encoded CBOR data item).

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

Less than 24: The argument's value is the value of the additional information.

24, 25, 26, or 27: The argument's value is held in the following 1, 2, 4, or 8 bytes, respectively, in network byte order. For major type 7 and additional information value 25, 26, 27, these bytes are not used as an integer argument, but as a floating-point value (see Section 3.3).

28, 29, 30: These values are reserved for future additions to the CBOR format. In the present version of CBOR, the encoded item is not well-formed.

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

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

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

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

A CBOR decoder implementation can be based on a jump table with all 256 defined values for the initial byte (Table 7). A decoder in a constrained implementation can instead use the structure of the initial byte and following bytes for more compact code (see Appendix C for a rough impression of how this could look).

3.1. Major Types

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

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

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

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

Major type 3: a text string (Section 2), encoded as UTF-8 ([RFC3629]). The number of bytes in the string is equal to the argument. A string containing an invalid UTF-8 sequence is well-formed but invalid (Section 1.2). This type is provided for systems that need to interpret or display human-readable text, and allows the differentiation between unstructured bytes and text that has a specified repertoire 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. In other formats, arrays are also called lists, sequences, or tuples (a "CBOR sequence" is something slightly different, though [RFC8742]). The argument is the number of data items in the array. Items in an array do not need to all be of the same type. For example, an array that contains 10 items of any type would have an initial byte of 0b100_01010 (major type of 4, additional information of 10 for the length) followed by the 10 remaining items.

Major type 5: a map of pairs of data items. Maps are also called tables, dictionaries, hashes, or objects (in JSON). A map is comprised of pairs of data items, each pair consisting of a key that is immediately followed by a value. The argument is the number of _pairs_ of data items in the map. For example, a map that contains 9 pairs would have an initial byte of 0b101_01001 (major type of 5, additional information of 9 for the number of pairs) followed by the 18 remaining items. The first item is the first key, the second item is the first value, the third item is the second key, and so on. Because items in a map come in pairs, their total number is always even: A map that contains an odd number of items (no value data present after the last key data item) is not well-formed. A map that has duplicate keys may be well-formed, but it is not valid, and thus it causes indeterminate decoding; see also Section 5.6.

Major type 6: a tagged data item ("tag") whose tag number, an integer in the range 0..2**64-1 inclusive, is the argument and whose enclosed data item ("tag content") is the single encoded data item that follows the head. See Section 3.4.

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

These eight major types lead to a simple table showing which of the 256 possible values for the initial byte of a data item are used (Table 7).

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

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

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

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

3.2. Indefinite Lengths for Some Major Types

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

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

3.2.1. The "break" Stop Code

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

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

3.2.2. Indefinite-Length Arrays and Maps

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

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

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

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

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

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

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

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

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

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

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

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

3.2.3. Indefinite-Length Byte Strings and Text Strings

Indefinite-length strings are represented by a byte containing the major type and additional information value of 31, followed by a series of zero or more byte or text strings ("chunks") that have definite lengths, followed by the "break" stop code (Section 3.2.1). The data item represented by the indefinite-length string is the concatenation of the chunks (i.e., the empty byte or text string, respectively, if no chunk is present). (Note that zero-length chunks, while not particularly useful, are permitted.)

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

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

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

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

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

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

3.2.4. Summary of indefinite-length use of major types

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

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

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

3.3. Floating-Point Numbers and Values with No Content

Major type 7 is for two types of data: floating-point numbers and "simple values" that do not need any content. Each value of the 5-bit additional information in the initial byte has its own separate meaning, as defined in Table 3. Like the major types for integers, items of this major type do not carry content data; all the information is in the initial bytes.

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

Table 3: Values for Additional Information in Major Type 7

As with all other major types, the 5-bit value 24 signifies a single-byte extension: it is followed by an additional byte to represent the simple value. (To minimize confusion, only the values 32 to 255 are used.) This maintains the structure of the initial bytes: as for the other major types, the length of these always depends on the additional information in the first byte. Table 4 lists the 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 4: Simple Values

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

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

3.4. Tagging of Items

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

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

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

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

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

Decoders do not need to understand tags of every tag number, and tags may be of little value in applications where the implementation creating a particular CBOR data item and the implementation decoding that stream know the semantic meaning of each item in the data flow. Their primary purpose in this specification is to define common data types such as dates. A secondary purpose is to provide conversion hints when it is foreseen that the CBOR data item needs to be translated into a different format, requiring hints about the content of items. Understanding the semantics of tags is optional for a decoder; it can simply present both the tag number and the tag content to the application, without interpreting the additional semantics of the tag.

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

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

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

35	text string	Regular expression; see Section 3.4.5.3
36	text string	MIME message; see Section 3.4.5.3
55799	(any)	Self-described CBOR; see Section 3.4.6

Table 5: Tag numbers defined in RFC 7049

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

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

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

3.4.1. Standard Date/Time String

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

3.4.2. Epoch-based Date/Time

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

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

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

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

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

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

3.4.3. Bignums

Protocols using tag numbers 2 and 3 extend the generic data model (Section 2) with "bignums" representing arbitrarily sized integers. In the basic generic data model, bignum values are not equal to integers from the same model, but the extended generic data model created by this tag definition defines equivalence based on numeric value, and preferred serialization (Section 4.1) never makes use of bignums that also can be expressed as basic integers (see below).

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

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

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

3.4.4. Decimal Fractions and Bigfloats

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

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

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

A decimal fraction or a bigfloat is represented as a tagged array that contains exactly two integer numbers: an exponent e and a mantissa m. Decimal fractions (tag number 4) use base-10 exponents; the value of a decimal fraction data item is m*(10**e). Bigfloats (tag number 5) use base-2 exponents; the value of a bigfloat data item is m*(2**e). The exponent e MUST be represented in an integer of major type 0 or 1, while the mantissa can also be a bignum (Section 3.4.3). Contained items with other structures are invalid.

An example of a decimal fraction is that the number 273.15 could be represented as 0b110_00100 (major type of 6 for the tag, additional information of 4 for the number of tag), followed by 0b100_00010 (major type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00001 (major type of 1 for the first integer, additional information of 1 for the value of -2), followed by 0b000_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 number of tag), followed by 0b100_00010 (major type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00000 (major type of 1 for the first integer, additional information of 0 for the value of -1), followed by 0b000_00011 (major type of 0 for the second integer, additional information of 3 for the value of 3). In hexadecimal:

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

Decimal fractions and bigfloats provide no representation of Infinity, -Infinity, or NaN; if these are needed in place of a decimal fraction or bigfloat, the IEEE 754 half-precision representations from Section 3.3 can be used.

3.4.5. Content Hints

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

3.4.5.1. Encoded CBOR Data Item

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

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

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

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

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

3.4.5.3. Encoded Text

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

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

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

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

  - the padding bits in a 2- or 3-character block are not 0, or

  - the base64 encoding has the wrong number of padding characters, or

  - the base64url encoding has padding characters,

  the string is invalid.

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

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

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

3.4.6. Self-Described CBOR

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

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

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

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

4. Serialization Considerations

4.1. Preferred Serialization

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

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

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

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

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

Definite length encoding is preferred whenever the length is known at the time the serialization of the item starts.

4.2. Deterministically Encoded CBOR

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

4.2.1. Core Deterministic Encoding Requirements

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

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

  - 0 to 23 and -1 to -24 MUST be expressed in the same byte as the major type;

  - 24 to 255 and -25 to -256 MUST be expressed only with an additional uint8_t;

  - 256 to 65535 and -257 to -65536 MUST be expressed only with an additional uint16_t;

  - 65536 to 4294967295 and -65537 to -4294967296 MUST be expressed only with an additional uint32_t.

  Floating-point values also MUST use the shortest form that preserves the value, e.g. 1.5 is encoded as 0xf93e00 and 1000000.5 as 0xfa49742408. (One implementation of this is to have all floats start as a 64-bit float, then do a test conversion to a 32-bit float; if the result is the same numeric value, use the shorter form and repeat the process with a test conversion to a 16-bit float. This also works to select 16-bit float for positive and negative Infinity as well.)

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

* The keys in every map MUST be sorted in the bytewise lexicographic order of their deterministic encodings. For example, the following keys are sorted correctly:

  1. 10, encoded as 0x0a.

  2. 100, encoded as 0x1864.

  3. -1, encoded as 0x20.

  4. "z", encoded as 0x617a.

  5. "aa", encoded as 0x626161.

  6. [100], encoded as 0x811864.

  7. [-1], encoded as 0x8120.

  8. false, encoded as 0xf4.

4.2.2. Additional Deterministic Encoding Considerations

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

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

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

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

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

* If a protocol includes a field that can express floating-point values, with a specific data model that declares integer and floating-point values to be interchangeable, the protocol's deterministic encoding needs to specify whether the integer 1.0 is encoded as 0x01, 0xf93c00, 0xfa3f800000, or 0xfb3ff0000000000000. Example rules for this are:

  1. Encode integral values that fit in 64 bits as values from major types 0 and 1, and other values as the preferred (smallest of 16-, 32-, or 64-bit) floating-point representation that accurately represents the value,

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

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

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

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

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

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

4.2.3. Length-first Map Key Ordering

The core deterministic encoding requirements (Section 4.2.1) sort map keys in a different order from the one suggested by Section 3.9 of [RFC7049] (called "Canonical CBOR" there). Protocols that need to be compatible with [RFC7049]'s order can instead be specified in terms of this specification's "length-first core deterministic encoding requirements":

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

1. If two keys have different lengths, the shorter one sorts earlier;

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

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

1. 10, encoded as 0x0a.

2. -1, encoded as 0x20.

3. false, encoded as 0xf4.

4. 100, encoded as 0x1864.

5. "z", encoded as 0x617a.

6. [-1], encoded as 0x8120.

7. "aa", encoded as 0x626161.

8. [100], encoded as 0x811864.

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

5. Creating CBOR-Based Protocols

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

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

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

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

5.1. CBOR in Streaming Applications

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

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

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

5.2. Generic Encoders and Decoders

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

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

Even though CBOR attempts to minimize these cases, not all well-formed CBOR data is valid: for example, the encoded text string "0x62c0ae" does not contain valid UTF-8 (because [RFC3629] requires always using the shortest form) and so is not a valid CBOR item. Also, specific tags may make semantic constraints that may be violated, for instance by a bignum tag enclosing another tag, or by an instance of tag number 0 containing a byte string, or containing a text string with contents that do not match [RFC3339]'s "date-time" production. There is no requirement that generic encoders and decoders make unnatural choices for their application interface to enable the processing of invalid data. Generic encoders and decoders are expected to forward simple values and tags even if their specific codepoints are not registered at the time the encoder/decoder is written (Section 5.4).

5.3. Validity of Items

A well-formed but invalid CBOR data item (Section 1.2) presents a problem with interpreting the data encoded in it in the CBOR data model. A CBOR-based protocol could be specified in several layers, in which the lower layers don't process the semantics of some of the CBOR data they forward. These layers can't notice any validity errors in data they don't process and MUST forward that data as-is. The first layer that does process the semantics of an invalid CBOR item MUST take one of two choices:

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

2. Issue an error and stop processing altogether.

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

Such problems might occur at the basic validity level of CBOR or in the context of tags (tag validity).

5.3.1. Basic validity

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

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

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

5.3.2. Tag validity

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

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

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

5.4. Validity and Evolution

A decoder with validity checking will expend the effort to reliably detect data items with validity errors. For example, such a decoder needs to have an API that reports an error (and does not return data) for a CBOR data item that contains any of the validity errors listed in the previous subsection.

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

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

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

The latter approach, which is also appropriate for decoders that do not support validity checking, provides forward compatibility with newly registered tags and simple values without the requirement to update the encoder at the same time as the calling application. (For this, the API for the decoder needs to have a way to mark unknown items so that the calling application can handle them in a manner appropriate for the program.)

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

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

5.5. Numbers

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

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

A CBOR-based protocol designed for compactness may want to exclude specific integer encodings that are longer than necessary for the application, such as to save the need to implement 64-bit integers. There is an expectation that encoders will use the most compact integer representation that can represent a given value. However, a compact application that does not require deterministic encoding should accept values that use a longer-than-needed encoding (such as encoding "0" as 0b000_11001 followed by two bytes of 0x00) as long as the application can decode an integer of the given size. Similar considerations apply to floating-point values; decoding both preferred serializations and longer-than-needed ones is recommended.

CBOR-based protocols for constrained applications that provide a choice between representing a specific number as an integer and as a decimal fraction or bigfloat (such as when the exponent is small and non-negative), might express a quality-of-implementation expectation that the integer representation is used directly.

5.6. Specifying Keys for Maps

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

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

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

A CBOR-based protocol MUST define what to do when a receiving application does see multiple identical keys in a map. The resulting rule in the protocol MUST respect the CBOR data model: it cannot prescribe a specific handling of the entries with the identical keys, except that it might have a rule that having identical keys in a map indicates a malformed map and that the decoder has to stop with an error. When processing maps that exhibit entries with duplicate keys, a generic decoder might do one of the following:

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

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

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

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

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

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

5.6.1. Equivalence of Keys

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

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

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

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

Tagged values are equal if both the tag number and the tag content are equal. (Note that a generic decoder that provides processing for a specific tag may not be able to distinguish some semantically equivalent values, e.g. if leading zeroes occur in the content of tag 2/3 (Section 3.4.3).) Simple values are equal if they simply have the same value. Nothing else is equal in the generic data model, a simple value 2 is not equivalent to an integer 2 and an array is never equivalent to a map.

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

5.7. Undefined Values

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

6. Converting Data between CBOR and JSON

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

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

6.1. Converting from CBOR to JSON

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

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

* A byte string (major type 2) that is not embedded in a tag that specifies a proposed encoding is encoded in base64url without padding and becomes a JSON string.

* A UTF-8 string (major type 3) becomes a JSON string. Note that JSON requires escaping certain characters ([RFC8259], Section 7): quotation mark (U+0022), reverse solidus (U+005C), and the "C0 control characters" (U+0000 through U+001F). All other characters are copied unchanged into the JSON UTF-8 string.

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

* A map (major type 5) becomes a JSON object. This is possible directly only if all keys are UTF-8 strings. A converter might also convert other keys into UTF-8 strings (such as by converting integers into strings containing their decimal representation); however, doing so introduces a danger of key collision. Note also that, if tags on UTF-8 strings are ignored as proposed below, this will cause a key collision if the tags are different but the strings are the same.

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

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

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

* A floating-point value (major type 7, additional information 25 through 27) becomes a JSON number if it is finite (that is, it can be represented in a JSON number); if the value is non-finite (NaN, or positive or negative Infinity), it is represented by the substitute value.

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

* A bignum (major type 6, tag number 2 or 3) is represented by encoding its byte string in base64url without padding and becomes a JSON string. For tag number 3 (negative bignum), a "~" (ASCII tilde) is inserted before the base-encoded value. (The conversion to a binary blob instead of a number is to prevent a likely numeric overflow for the JSON decoder.)

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

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

* Indefinite-length items are made definite before conversion.

6.2. Converting from JSON to CBOR

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

* JSON numbers without fractional parts (integer numbers) are represented as integers (major types 0 and 1, possibly major type 6 tag number 2 and 3), choosing the shortest form; integers longer than an implementation-defined threshold may instead be represented as floating-point values. The default range that is represented as integer is -2**53+1..2**53-1 (fully exploiting the range for exact integers in the binary64 representation often used for decoding JSON [RFC7493]). A CBOR-based protocol, or a generic

converter implementation, may choose -2**32..2**32-1 or -2**64..2**64-1 (fully using the integer ranges available in CBOR with uint32_t or uint64_t, respectively) or even -2**31..2**31-1 or -2**63..2**63-1 (using popular ranges for two's complement signed integers). (If the JSON was generated from a JavaScript implementation, its precision is already limited to 53 bits maximum.)

* Numbers with fractional parts are represented as floating-point values, performing the decimal-to-binary conversion based on the precision provided by IEEE 754 binary64. Then, when encoding in CBOR, the preferred serialization uses the shortest floating-point representation exactly representing this conversion result; for instance, 1.5 is represented in a 16-bit floating-point value (not all implementations will be capable of efficiently finding the minimum form, though). Instead of using the default binary64 precision, there may be an implementation-defined limit to the precision of the conversion that will affect the precision of the represented values. Decimal representation should only be used on the CBOR side if that is specified in a protocol.

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

7. Future Evolution of CBOR

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

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

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

7.1. Extension Points

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

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

CBOR has three major extension points:

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

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

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

7.2. Curating the Additional Information Space

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

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

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

8. Diagnostic Notation

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

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

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

The notation borrows the JSON syntax for numbers (integer and floating-point), True (>true<), False (>false<), Null (>null<), UTF-8 strings, arrays, and maps (maps are called objects in JSON; the diagnostic notation extends JSON here by allowing any data item in the key position). Undefined is written >undefined< as in

JavaScript. The non-finite floating-point numbers Infinity, -Infinity, and NaN are written exactly as in this sentence (this is also a way they can be written in JavaScript, although JSON does not allow them). A tag is written as an integer number for the tag number, followed by the tag content in parentheses; for instance, an RFC 3339 (ISO 8601) date could be notated as:

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

or the equivalent relative time as

```
1(1363896240)
```

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

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

A number of useful extensions to the diagnostic notation defined here are provided in Appendix G of [RFC8610], "Extended Diagnostic Notation" (EDN).

8.1. Encoding Indicators

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

The convention for encoding indicators is that anything starting with an underscore and all following characters that are alphanumeric or underscore, is an encoding indicator, and can be ignored by anyone not interested in this information. For example, "_" or "_3". Encoding indicators are always optional.

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

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

Byte and text strings of indefinite length can be notated in the form (_ h'0123', h'4567') and (_ "foo", "bar").

9. IANA Considerations

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

[To be removed by RFC editor:] IANA is requested to update these registries to point to the present document instead of RFC 7049.

9.1. Simple Values Registry

IANA has created the "Concise Binary Object Representation (CBOR) Simple Values" registry at [IANA.cbor-simple-values]. The initial values are shown in Table 4.

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

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

9.2. Tags Registry

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

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

* Data item

* Semantics (short form)

In addition, First Come First Served requests should include:

* Point of contact

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

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

9.3. Media Type ("MIME Type")

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

Type name: application

Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 10 of this document

Interoperability considerations: n/a

Published specification: This document

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

Intended usage: COMMON

Restrictions on usage: none

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

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

9.4. CoAP Content-Format

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

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]

9.5. The +cbor Structured Syntax Suffix Registration

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

Name: Concise Binary Object Representation (CBOR)

+suffix: +cbor

References: [RFCthis]

Encoding Considerations: CBOR is a binary format.

Interoperability Considerations: n/a

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

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

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

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

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

Security Considerations: See Section 10 of this document

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

Author/Change Controller: The IESG iesg@ietf.org (mailto:iesg@ietf.org)
// Editors’ note: RFC 6838 has a template field Author/Change
// controller, the descriptive text of which makes clear that this
is
// the change controller, not the author. Go figure. There is no
// separate author entry as in the media types registry. (RFC
// editor: Please remove this note before publication.)

10. Security Considerations

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

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

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

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

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

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

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

Protocols should be defined in such a way that potential multiple interpretations are reliably reduced to a single interpretation. For example, an attacker could make use of invalid input such as duplicate keys in maps, or exploit different precision in processing numbers to make one application base its decisions on a different interpretation than the one that will be used by a second application. To facilitate consistent interpretation, encoder and decoder implementations should provide a validity checking mode of operation (Section 5.4). Note, however, that a generic decoder cannot know about all requirements that an application poses on its input data; it is therefore not relieving the application from performing its own input checking. Also, since the set of defined tag numbers evolves, the application may employ a tag number that is not yet supported for validity checking by the generic decoder it uses. Generic decoders therefore need to provide documentation which tag numbers they support and what validity checking they can provide for each of them as well as for basic CBOR validity (UTF-8 checking, duplicate map key checking).

11. References

11.1. Normative References

[ECMA262] Ecma International, "ECMAScript 2018 Language Specification", ECMA Standard ECMA-262, 9th Edition, June 2018, <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2008.

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

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

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

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

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, DOI 10.17487/RFC4287, December 2005, <https://www.rfc-editor.org/info/rfc4287>.

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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

11.2. Informative References

[ASN.1] International Telecommunication Union, "Information Technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.

[BSON] Various, "BSON - Binary JSON", 2013, <http://bsonspec.org/>.

[I-D.bormann-cbor-notable-tags] Bormann, C., "Notable CBOR Tags", Work in Progress, Internet-Draft, draft-bormann-cbor-notable-tags-01, 15 May 2020, <http://www.ietf.org/internet-drafts/draft-bormann-cbor-notable-tags-01.txt>.

[IANA.cbor-simple-values] IANA, "Concise Binary Object Representation (CBOR) Simple Values", <http://www.iana.org/assignments/cbor-simple-values>.

[IANA.cbor-tags] IANA, "Concise Binary Object Representation (CBOR) Tags", <http://www.iana.org/assignments/cbor-tags>.

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

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

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

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

[PCRE] Ho, A., "PCRE - Perl Compatible Regular Expressions", 2018, <http://www.pcre.org/>.

[RFC0713] Haverty, J., "MSDTP-Message Services Data Transmission Protocol", RFC 713, DOI 10.17487/RFC0713, April 1976, <https://www.rfc-editor.org/info/rfc713>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

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

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

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

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

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

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

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

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

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

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

[YAML] Ben-Kiki, O., Evans, C., and I.d. Net, "YAML Ain't Markup Language (YAML[TM]) Version 1.2", 3rd Edition, October 2009, <http://www.yaml.org/spec/1.2/spec.html>.

Appendix A. Examples

The following table provides some CBOR-encoded values in hexadecimal (right column), together with diagnostic notation for these values (left column). Note that the string "\u00fc" is one form of diagnostic notation for a UTF-8 string containing the single Unicode character U+00FC, LATIN SMALL LETTER U WITH DIAERESIS (u umlaut). Similarly, "\u6c34" is a UTF-8 string in diagnostic notation with a single character U+6C34 (CJK UNIFIED IDEOGRAPH-6C34, often representing "water"), and "\ud800\udd51" is a UTF-8 string in diagnostic notation with a single character U+10151 (GREEK ACROPHONIC ATTIC FIFTY STATERS). (Note that all these single-character strings could also be represented in native UTF-8 in diagnostic notation, just not in an ASCII-only specification like the present one.) In the diagnostic notation provided for bignums, their intended numeric value is shown as a decimal number (such as 18446744073709551616) instead of showing a tagged byte string (such as 2(h'010000000000000000')).

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

18446744073709551616	0xc249010000000000000000
-18446744073709551616	0x3bffffffffffffffff
-18446744073709551617	0xc349010000000000000000
-1	0x20
-10	0x29
-100	0x3863
-1000	0x3903e7
0.0	0xf90000
-0.0	0xf98000
1.0	0xf93c00
1.1	0xfb3ff199999999999a
1.5	0xf93e00
65504.0	0xf97bff
100000.0	0xfa47c35000
3.4028234663852886e+38	0xfa7f7fffff
1.0e+300	0xfb7e37e43c8800759c
5.960464477539063e-8	0xf90001
0.00006103515625	0xf90400
-4.0	0xf9c400
-4.1	0xfbc010666666666666
Infinity	0xf97c00
NaN	0xf97e00
-Infinity	0xf9fc00
Infinity	0xfa7f800000

NaN	0xfa7fc00000
-Infinity	0xfaff800000
Infinity	0xfb7ff0000000000000
NaN	0xfb7ff8000000000000
-Infinity	0xfbfff0000000000000
false	0xf4
true	0xf5
null	0xf6
undefined	0xf7
simple(16)	0xf0
simple(255)	0xf8ff
0("2013-03-21T20:04:00Z")	0xc074323031332d30332d32315432303a 30343a30305a
1(1363896240)	0xc11a514b67b0
1(1363896240.5)	0xc1fb41d452d9ec200000
23(h'01020304')	0xd74401020304
24(h'6449455446')	0xd818456449455446
32("http://www.example.com")	0xd82076687474703a2f2f7777772e6578 616d706c652e636f6d
h''	0x40
h'01020304'	0x4401020304
""	0x60
"a"	0x6161
"IETF"	0x6449455446
"\"\\"	0x62225c

"\u00fc"	0x62c3bc
"\u6c34"	0x63e6b0b4
"\ud800\udd51"	0x64f0908591
[]	0x80
[1, 2, 3]	0x83010203
[1, [2, 3], [4, 5]]	0x8301820203820405
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	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"}	0xa5616161416162614261636143616461 4461656145
(_ h'0102', h'030405')	0x5f42010243030405ff
(_ "strea", "ming")	0x7f657374726561646d696e67ff
[_]	0x9fff
[_ 1, [2, 3], [_ 4, 5]]	0x9f018202039f0405ffff
[_ 1, [2, 3], [4, 5]]	0x9f01820203820405ff
[1, [2, 3], [_ 4, 5]]	0x83018202039f0405ff
[1, [_ 2, 3], [4, 5]]	0x83019f0203ff820405
[_ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x9f0102030405060708090a0b0c0d0e0f101112131415161718181819ff
{_ "a": 1, "b": [_ 2, 3]}	0xbf61610161629f0203ffff

["a", {_ "b": "c"}]	0x826161bf61626163ff
{_ "Fun": true, "Amt": -2}	0xbf6346756ef563416d7421ff

Table 6: Examples of Encoded CBOR Data Items

Appendix B. Jump Table

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

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

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

	of data items follow)
0xb9	map (two-byte uint16_t for n, and then n pairs of data items follow)
0xba	map (four-byte uint32_t for n, and then n pairs of data items follow)
0xbb	map (eight-byte uint64_t for n, and then n pairs of data items follow)
0xbf	map, pairs of data items follow, terminated by "break"
0xc0	Text-based date/time (data item follows; see Section 3.4.1)
0xc1	Epoch-based date/time (data item follows; see Section 3.4.2)
0xc2	Positive bignum (data item "byte string" follows)
0xc3	Negative bignum (data item "byte string" follows)
0xc4	Decimal Fraction (data item "array" follows; see Section 3.4.4)
0xc5	Bigfloat (data item "array" follows; see Section 3.4.4)
0xc6..0xd4	(tag)
0xd5..0xd7	Expected Conversion (data item follows; see Section 3.4.5.2)
0xd8..0xdb	(more tags, 1/2/4/8 bytes and then a data item follow)
0xe0..0xf3	(simple value)
0xf4	False
0xf5	True
0xf6	Null

0xf7	Undefined
0xf8	(simple value, one byte follows)
0xf9	Half-Precision Float (two-byte IEEE 754)
0xfa	Single-Precision Float (four-byte IEEE 754)
0xfb	Double-Precision Float (eight-byte IEEE 754)
0xff	"break" stop code

Table 7: Jump Table for Initial Byte

Appendix C. Pseudocode

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

* the pseudocode does not "fail";

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

The pseudocode has the following prerequisites:

* take(n) reads n bytes from the input data and returns them as a byte string. If n bytes are no longer available, take(n) fails.

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

* Arithmetic works as in C.

* All variables are unsigned integers of sufficient range.

Note that "well_formed" returns the major type for well-formed definite length items, but 0 for an indefinite length item (or -1 for a "break" stop code, only if "breakable" is set). This is used in "well_formed_indefinite" to ascertain that indefinite length strings only contain definite length strings as chunks.

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

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

Figure 1: Pseudocode for Well-Formedness Check

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

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

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

Figure 2: Pseudocode for Encoding a Signed Integer

Appendix D. Half-Precision

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

```
#include <math.h>

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

Figure 3: C Code for a Half-Precision Decoder

```
import struct
from math import ldexp

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

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

Figure 4: Python Code for a Half-Precision Decoder

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

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

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

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

1. unambiguous encoding of most common data formats from Internet standards

2. code compactness for encoder or decoder

3. no schema description needed

4. reasonably compact serialization

5. applicability to constrained and unconstrained applications

6. good JSON conversion

7. extensibility

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

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

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

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

E.2. MessagePack

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

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

E.3. BSON

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

E.4. MSDTP: RFC 713

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

E.5. Conciseness on the Wire

While CBOR's design objective of code compactness for encoders and decoders is a higher priority than its objective of conciseness on the wire, many people focus on the wire size. Table 8 shows some encoding examples for the simple nested array [1, [2, 3]]; where some form of indefinite-length encoding is supported by the encoding, [_ 1, [2, 3]] (indefinite length on the outer array) is also shown.

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

Table 8: Examples for Different Levels of Conciseness

Appendix F. Well-formedness errors and examples

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

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

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

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

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

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

* major type 7, additional information 24, value < 32 (incorrect or incorrectly encoded simple type)

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

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

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

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

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

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

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

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

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

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

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

* Tag number not followed by tag content: c0

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

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

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

Subkind 1:

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

Subkind 2:

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

Subkind 3:

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

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

Subkind 4:

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

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

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

Subkind 5:

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

Appendix G. Changes from RFC 7049

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

G.1. Errata processing, clerical changes

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

Other more clerical changes include:

* use of new RFCXML functionality [RFC7991];

* explain some more of the notation used;

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

* the discussion of diagnostic notation mentions the "Extended Diagnostic Notation" (EDN) defined in [RFC8610];

* the addition of this appendix.

G.2. Changes in IANA considerations

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

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

G.3. Changes in suggestions and other informational components

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

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

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

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

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

The handling of non-well-formed simple values was clarified in text and pseudocode. Appendix F was added to discuss well-formedness errors and provide examples for them.

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

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

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

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

Acknowledgements

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

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

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

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

Authors' Addresses

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

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

Paul Hoffman
ICANN

Email: paul.hoffman@icann.org

CBOR Working Group M. Jones
Internet-Draft A. Nadalin
Intended status: Standards Track Microsoft
Expires: January 17, 2021 J. Richter
pdv Financial Software GmbH
July 16, 2020

Concise Binary Object Representation (CBOR) Tags for Date
draft-ietf-cbor-date-tag-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.

In CBOR, one point of extensibility is the definition of CBOR tags. RFC 7049 defines two tags for time: CBOR tag 0 (RFC 3339 date/time string) and tag 1 (Posix "seconds since the epoch"). Since then, additional requirements have become known. This specification defines a CBOR tag for an RFC 3339 date text string, for applications needing a textual date representation within the Gregorian calendar without a time. It also defines a CBOR tag for days since the date 1970-01-01 in the Gregorian calendar for applications needing a numeric date representation without a time. This specification is intended as the reference document for IANA registration of the CBOR tags defined.

Status of This Memo

Table of Contents

1. Introduction

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

This specification defines a CBOR tag for a text string representing a date without a time. The tagged text string is represented as specified by the RFC 3339 [RFC3339] "full-date" production. Per RFC 3339, this represents a date within the Gregorian calendar.

This specification also defines a CBOR tag for an integer representing a date without a time. The tagged integer is an unsigned or negative value indicating the number of days since the Gregorian calendar date 1970-01-01. As an implementation note, this value has a constant offset from the Modified Julian Date value (which is defined by the Smithsonian Astrophysical Observatory as the number of days since November 17, 1858); this value is the Modified Julian Date minus 40587.

Note that since both tags are for dates without times, times of day, time zones, and leap seconds are not applicable to these values. These tags are both for representations of Gregorian calendar dates.

1.1. Calendar Dates

Calendar dates are used for numerous human use cases, such as marking the dates of significant events. For instance, John Lennon was born on October 9, 1940 and died on December 8, 1980. One such use case is driver's licenses, which typically include a date of birth. The dates used in this specification use the Gregorian calendar, as do those in RFC 3339 [RFC3339]. The time zones and actual times of these events are intentionally not represented in the calendar date.

The epoch chosen for the second tag, which represents days since the Gregorian calendar date 1970-01-01, is related to the IEEE Std 1003.1, 2013 Edition [POSIX.1] time epoch 1970-01-01T00:00:00Z UTC only insofar as both contain the date 1970-01-01. This should not be construed as indicating that dates using this tag represent either a specific time of day and/or time zone.

The day of the week (Sunday, Monday, Tuesday, etc.) is not explicitly represented in either of these date formats. However, deterministic algorithms that are beyond the scope of this specification can be used to derive the day of the week in the Gregorian calendar from dates represented in both of these formats.

1.1.1. Example Date Representations

This table contains example representations for dates using both tags.

Date	Tag 1004	Tag 100
October 9, 1940	"1940-10-09"	-10676
December 8, 1980	"1980-12-08"	3994

1.2. Comparing Dates

Comparison of dates in "full-date" format can be accomplished by normal string comparison, since by design, the digits representing the date are in fixed format and ordered from most significant to least significant. Comparison of numeric dates representing days since 1970-01-01 can be performed by normal integer comparison. Comparison of dates in other formats or using other calendars require conversions that are beyond the scope of this specification.

Note that different dates may correspond to the same moment in time, depending upon the time zone in which the date was determined. For instance, at many times of the day, a conference call occurring on a particular date in Japan will simultaneously occur on the previous date in Hawaii; at many times of the day, Japan's Friday corresponds with Hawaii's Thursday.

1.3. Comparing Dates and Date/Time Values

Comparing dates with date/time values, which represent a particular moment in time, is beyond the scope of this specification. That said, if a date is augmented with a time zone and time of day, a specific date/time value can be determined and comparing that date/time value to others becomes possible. For instance, if one were to augment John Lennon's birth date of October 9, 1940 with the time of day and time zone of his birth, then it would be possible to derive a date/time at which he was born that could be compared with other date/time values.

2. IANA Considerations

2.1. Concise Binary Object Representation (CBOR) Tags Registrations

This section registers the following values in the IANA "Concise Binary Object Representation (CBOR) Tags" registry [IANA.cbor-tags].

- o Tag: 1004
- o Data Item: UTF-8 text string
- o Semantics: RFC 3339 full-date string
- o Reference: [[this specification]]

- o Tag: 100 (ASCII 'd')
- o Data Item: Unsigned or negative integer
- o Semantics: Number of days since the epoch date 1970-01-01
- o Reference: [[this specification]]

3. Security Considerations

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

A date, of course, has significant security considerations. These include the exploitation of ambiguities where the date is security relevant or where the date is used in access control decisions.

When using a calendar date for decision making, for example access control, it needs to be noted that since calendar dates do not represent a specific point in time, the results of the evaluation can differ depending upon where the decision is made. For instance, a person may have reached their 21st birthday in Japan while simultaneously being a day short of their 21st birthday in Hawaii.

4. References

4.1. Normative References

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

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

4.2. Informative References

[IANA.cbor-tags] IANA, "Concise Binary Object Representation (CBOR) Tags", <http://www.iana.org/assignments/cbor-tags>.

[POSIX.1] IEEE, "The Open Group Base Specifications Issue 7", IEEE Std 1003.1, 2013 Edition, 2013, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15>.

Acknowledgements

Thanks to Carsten Bormann for supporting creation of this specification. Parts of the explanatory text in this specification come from draft-bormann-cbor-time-tag-02.

Thanks to these people for reviews of the specification: Henk Birkholz, Carsten Bormann, Thiago Macieira, Francesca Palombini,

Michael Richardson, Jim Schaad, Juergen Schoenwaelder, and Dale Worley.

Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-05

- o Incorporated additional suggestions by Carsten Bormann and Juergen Schoenwaelder.

-04

- o Addressed shepherd comments by Francesca Palombini.
- o Addressed additional review comments by Jim Schaad and Michael Richardson.

-03

- o Added statement that these tags are both for representations of calendar dates.
- o Described consequences of using calendar dates in access control decisions.

-02

- o Addressed working group last call comments, including stating that time zones are not applicable to these values.

-01

- o Changed "positive or negative" to "unsigned or negative".
- o Added an implementation note about the relationship to Modified Julian Dates.

-00

- o Initial working group version based on draft-jones-cbor-date-tag-01 with no normative changes.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: https://self-issued.info/

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

Joerg Richter
pdv Financial Software GmbH

Email: joerg.richter@pdv-fs.de