The CDE-based Application Profile dCBOR
draft-bormann-cbor-dcbor-04

Abstract

CBOR (STD 94, RFC 8949) defines "Deterministically Encoded CBOR" in its Section 4.2, providing some flexibility for application specific decisions. The CBOR Common Deterministic Encoding (CDE) Profile provides a more detailed common base for Deterministic Encoding, facilitating it be offered as a selectable feature of generic encoders, as well as the concept of Application Profiles that are layered on top of CDE. This document defines the application profile "dCBOR" as an example of such an application profile.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-bormann-cbor-dcbor/.

Discussion of this document takes place on the Concise Binary Object Representation Maintenance and Extensions (CBOR) Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cabo/det.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 May 2024.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction .................................................. 3
   1.1. Conventions and Definitions ........................... 3
2. Gordian dCBOR ................................................... 3
   2.1. Removing Simple Values ................................ 3
   2.2. Removing Integer Values ................................ 4
   2.3. Numeric Reduction of Floating-Point Values ............ 4
3. Extensibility ................................................... 5
   3.1. dCBOR-wide1 ............................................. 5
   3.2. dCBOR-wide2 ............................................. 7
4. CDDL support ..................................................... 9
5. Implementation Status .......................................... 9
   5.1. Gordian dCBOR Application Profile ....................... 10
      5.1.1. TypeScript ........................................... 10
      5.1.2. Swift ............................................... 10
      5.1.3. Rust ............................................... 10
      5.1.4. Ruby ............................................... 11
6. Security Considerations ........................................ 11
7. IANA Considerations ........................................... 11
8. References ..................................................... 11
   8.1. Normative References .................................. 11
   8.2. Informative References ................................. 12
Acknowledgments ................................................... 14
Contributors ...................................................... 14
Author’s Address .................................................. 14
1. Introduction

CBOR (STD 94, RFC 8949) defines "Deterministically Encoded CBOR" in its Section 4.2, providing some flexibility for application specific decisions. The CBOR Common Deterministic Encoding (CDE) Profile provides a more detailed common base for Deterministic Encoding, facilitating it be offered as a selectable feature of generic encoders, as well as the concept of Application Profiles that are layered on top of CDE. This document defines the application profile "dCBOR" as an example of such an application profile.

1.1. Conventions and Definitions

The definitions of [STD94] and the Common Deterministic Encoding (CDE) Profile [I-D.bormann-cbor-cde] apply.

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] when, and only when, they appear in all capitals, as shown here.

2. Gordian dCBOR

Gordian dCBOR [I-D.mcnally-deterministic-cbor] provides an application profile that requires encoders to produce valid CBOR in deterministic encoding as defined in CDE). Gordian dCBOR also requires dCBOR decoders to reject CBOR data items that were not deterministically encoded.

Beyond CDE, dCBOR imposes certain limitations on the CBOR basic generic data model. Some items that can be represented in the CBOR basic generic data model are entirely outlawed by this application profile. Other items are represented by what are considered equivalent data items by the dCBOR equivalence model, so a recipient application might receive data that may not be the same data in the CBOR equivalence model as the ones the generating application produced.

These restrictions mainly are about numeric values, which are therefore the subject of the main subsection of this section.

2.1. Removing Simple Values

Only the three simple values false (0xf4), true (0xf5), and null (0xf6) are allowed at the application level; the remaining 253 values must be rejected.
2.2. Removing Integer Values

Only the integer values in range \([-2^{63}, 2^{64}-1]\) can be expressed in dCBOR ("basic dCBOR integers"). Note that the range is asymmetric, with only \(2^{63}\) negative values, but \(2^{64}\) unsigned (non-negative) values, creating an (approximately) 64.6 bit integer.

This maps to a choice between a platform 64-bit two’s complement signed integer (often called int64) and a 64-bit unsigned integer (uint64). (Specific applications will, of course, further restrict ranges of integers that are considered valid for the application, based on their position and semantics in the CBOR data item.)

2.3. Numeric Reduction of Floating-Point Values

dCBOR implementations that do support floating point numbers MUST perform the following two reductions of numeric values when constructing CBOR data items:

1. When representing integral floating point values (floating point values with a zero fractional part), check whether the mathematically identical value can be represented as a dCBOR integer value, i.e., is in the range \([-2^{63}, 2^{64}-1]\) given above. If that is the case, convert the integral floating point to that mathematically identical integer value before encoding it. (Deterministic Encoding will then ensure the shortest length encoding is used.) This means that if a floating point value has a non-zero fractional part, or an exponent that takes it out of the given range of basic dCBOR integers, the original floating point value is used for encoding. (Specifically, conversion to a bignum is never considered.)

This also means that the three representations of a zero number in CBOR \((0, 0.0, -0.0\) in diagnostic notation) are all reduced to the basic integer 0 (with preferred encoding 0x00).

Note that this reduction can turn valid maps into invalid ones, as it can create duplicate keys, e.g., for:

```{ 
  10: "integer ten",
  10.0: "floating ten"
}
```

This means that, at the application level, the application MUST prevent the creation of maps that would turn invalid in dCBOR processing.
2. In addition, before encoding, represent all NaN values by using the quiet NaN value having the half-width CBOR representation 0xf97e00.

dCBOR-based applications MUST accept these "reduced" numbers in place of the original value, e.g., a dCBOR-based application that expects a floating point value needs to accept a basic dCBOR integer in its place (and, if needed, convert it to a floating point value for its own further processing).

dCBOR-based applications MUST NOT accept numbers that have not been reduced as specified in this section, except maybe by making the unreduced numbers available for their diagnostic value when there has been an explicit request to do so. This is similar to a checking flag mentioned in Section 5.1 (API Considerations) of [I-D.bormann-cbor-det] being set by default.

3. Extensibility

[I-D.mcnally-deterministic-cbor] does not discuss extensibility. A meaningful way to handle extensibility in this application profile would be to lift value range restrictions, keeping the profile-specific equivalence rules shown here intact and possibly adding equivalences as needed for newly allowed values.

This subsection presents two speculative extensions of dCBOR, called dCBOR-wide1 and dCBOR-wide2, to point out different objectives that can lead the development of an extension.

3.1. dCBOR-wide1

This speculative extension of dCBOR attempts to meet two objectives:

1. All instances that meet dCBOR are also instances of dCBOR-wide1; due to the nature of deterministic serialization this also means that dCBOR-wide1 instances that only use application data model values that are allowed by dCBOR are also dCBOR instances.

2. The range of integers that can be provided by an application and can be interchanged as exact numbers is expanded to \([-2^{127}, 2^{128}-1]\), now also covering the types i128 and u128 in Rust [i128][u128].

This extension is achieved by simply removing the integers in the extended range from the exclusion range of dCBOR. The numeric reduction rule is not changed, so it still applies only to integral-valued floating-point numbers in the range \([-2^{63}, 2^{64}-1]\).
Examples for the application-to-CDE mapping of dCBOR-wide1 are shown in Table 1. In the dCBOR column, items that are not excluded in dCBOR are marked , items that are excluded in dCBOR and therefore are new in dCBOR-wide1 are marked .

<table>
<thead>
<tr>
<th>Application data</th>
<th>dCBOR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding via CDE</td>
<td></td>
</tr>
<tr>
<td>Numeric reduction (if any)</td>
<td></td>
</tr>
</tbody>
</table>

| 0 |       |
| 00 |       |

| 0.0 |       |
| 0 |       |
| 00 |       |

| -0.0 |       |
| 0 |       |
| 00 |       |

| 4.0 |       |
| 4 |       |
| 04 |       |

| -4.0 |       |
| -4 |       |
| 23 |       |

| 1.0e+19 |       |
| 10000000000000000000 |       |
| 1B8AC7230489E80000 |       |

| -1.0e+19 |       |
| FBC3E158E460913D00 |       |

| 1.0e+38 |       |

| 1.0e+38 |       |
Table 1: Speculative "dCBOR-wide1" application profile

This speculative extended profile does not meet a potential objective number 3 that unextended dCBOR does meet:

3. All integral-valued floating point numbers coming from an application that fit into an integer representation allowed by the application profile are represented as such.

Objective 1 prevents numeric reduction from being applied to values that are not excluded in dCBOR but do to receive numeric reduction there.

3.2. dCBOR-wide2

The speculative dCBOR-wide2 extension of dCBOR attempts to meet objectives 2 and 3 mentioned in Section 3.1. It cannot meet objective 1: items in Table 2 marked with a character are allows in dCBOR but have different serializations.
Table 2: Speculative "dCBOR-wide2" application profile
This extension is achieved by removing the integers in the extended range from the exclusion range of dCBOR, and by adding the extended range to the target range of numeric reduction.

4. CDDL support

Similar to the CDDL [RFC8610] support in [I-D.bormann-cbor-cde], this specification adds two CDDL control operators that can be used to specify that the data items should be encoded in CBOR Common Deterministic Encoding (CDE), with the dCBOR application profile applied as well.

The control operators .dcbor and .dcborseq are exactly like .cde and .cdeseq except that they also require the encoded data item(s) to conform to the dCBOR application profile.

For example, the normative comment in Section 3 of [I-D.draft-mcnally-envelope-03]:

leaf = #6.24(bytes) ; MUST be dCBOR

...can now be formalized as:

leaf = #6.24(bytes .dcbor any)

5. Implementation Status

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

(Boilerplate as per Section 2.1 of [RFC7942]:)

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.
According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

5.1. Gordian dCBOR Application Profile

5.1.1. TypeScript

* Implementation Location: [bc-dcbor-ts]
* Primary Maintainer:
* Languages: TypeScript (transpiles to JavaScript)
* Coverage:
* Testing:
* Licensing:

5.1.2. Swift

* Implementation Location: [BCSwiftDCBOR]
* Primary Maintainer:
* Languages: Swift
* Coverage:
* Testing:
* Licensing: BSD-2-Clause-Patent

5.1.3. Rust

* Implementation Location: [bc-dcbor-rust]
* Primary Maintainer:
* Languages: Rust
* Coverage:
* Testing:
5.1.4. Ruby

* Implementation Location: [cbor-dcbor]
* Primary Maintainer: Carsten Bormann
* Languages: Ruby

* Coverage: Complete specification; complemented by CBOR encoder/decoder and command line interface from [cbor-diag] and deterministic encoding from [cbor-deterministic]. Checking of dCBOR exclusions not yet implemented.

* Testing: Also available at https://cbor.me (https://cbor.me)
* Licensing: Apache-2.0

6. Security Considerations

TODO Security

7. IANA Considerations

// RFC Editor: please replace RFCXXXX with the RFC number of this RFC // and remove this note.

This document requests IANA to register the contents of Table 3 into the registry "CDDL Control Operators" of [IANA.cddl]:

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dcbor</td>
<td>[RFCXXXX]</td>
</tr>
<tr>
<td>.dcborseq</td>
<td>[RFCXXXX]</td>
</tr>
</tbody>
</table>

Table 3: New control operators to be registered

8. References

8.1. Normative References
IANA, "Concise Data Definition Language (CDDL)",
<https://www.iana.org/assignments/cddl>.

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174,
DOI 10.17487/RFC8174, May 2017,

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,

Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949,
DOI 10.17487/RFC8949, December 2020,

8.2. Informative References

"Blockchain Commons Deterministic CBOR ("dCBOR") for Rust", n.d.,
<https://github.com/BlockchainCommons/bc-dcbor-rust>.

"Blockchain Commons Deterministic CBOR ("dCBOR") for TypeScript", n.d.,

"Blockchain Commons Deterministic CBOR ("dCBOR") for Swift", n.d.,
<https://github.com/BlockchainCommons/BCSwiftDCBOR>.

Bormann, C., "PoC of the McNally/Allen "dCBOR" application-level CBOR representation rules", n.d.,


Acknowledgments

This document is based on the work of Wolf McNally and Christopher Allen as documented in [I-D.mcnally-deterministic-cbor] and discussed in 2023 in the CBOR working group.

Contributors

Wolf McNally
Blockchain Commons
Email: wolf@wolfmcnally.com

Christopher Allen
Blockchain Commons
Email: christophera@lifewithalacrity.com

Author’s Address

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org
Abstract

The Concise Data Definition Language (CDDL), standardized in RFC 8610, provides "control operators" as its main language extension point. RFCs have added to this extension point both in an application-specific and a more general way.

The present document defines a number of additional generally applicable control operators for text conversion (Bytes, Integers, JSON, Printf-style formatting) and for an operation on text.

About This Document

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

The latest revision of this draft can be found at https://cbor-wg.github.io/cddl-more-control/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-cbor-cddl-more-control/.

Discussion of this document takes place on the Concise Binary Object Representation (CBOR) Maintenance and Extensions Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cbor-wg/cddl-more-control.

Status of This Memo

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

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

The Concise Data Definition Language (CDDL), standardized in [RFC8610], provides "control operators" as its main language extension point (Section 3.8 of [RFC8610]). RFCs have added to this extension point both in an application-specific [RFC9090] and a more general [RFC9165] way.
The present document defines a number of additional generally applicable control operators:

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>.b64u, .b64c</td>
<td>Base64 representation of byte strings</td>
</tr>
<tr>
<td>.b64u-sloppy,</td>
<td>(sloppy-tolerant variants of the</td>
</tr>
<tr>
<td>.b64c-sloppy</td>
<td>above)</td>
</tr>
<tr>
<td>.hex, .hexlc,</td>
<td>Base16 representation of byte strings</td>
</tr>
<tr>
<td>.hexuc</td>
<td></td>
</tr>
<tr>
<td>.b32, .h32</td>
<td>Base32 representation of byte strings</td>
</tr>
<tr>
<td>.b45</td>
<td>Base45 representation of byte strings</td>
</tr>
<tr>
<td>.decimal</td>
<td>Text representation of integer</td>
</tr>
<tr>
<td></td>
<td>numbers</td>
</tr>
<tr>
<td>.printf</td>
<td>Printf-formatted text representation</td>
</tr>
<tr>
<td></td>
<td>of data items</td>
</tr>
<tr>
<td>.json</td>
<td>Text representation of JSON values</td>
</tr>
<tr>
<td>.join</td>
<td>Building text from array of components</td>
</tr>
</tbody>
</table>

Table 1: New control operators in this document

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 [BCP14] when, and only when, they appear in all capitals, as shown here.

Regular expressions mentioned in the text are as defined in [RFC9485].
This specification uses terminology from [RFC8610]. In particular, with respect to control operators, "target" refers to the left-hand side operand, and "controller" to the right-hand side operand. "Tool" refers to tools along the lines of that described in Appendix F of [RFC8610]. Note also that the data model underlying CDDL provides for text strings as well as byte strings as two separate types, which are then collectively referred to as "strings".

2. Text Conversion

2.1. Byte Strings: Base16 (Hex), Base32, Base45, Base64

A CDDL model often defines data that are byte strings in essence but need to be transported in various encoded forms, such as base64 or hex. This section defines a number of control operators to model these conversions.

The control operators generally are of a form that could be used like this:

```
signature-for-json = text .b64u signature
signature = bytes .cbor COSE_Sign1
```

The specification of these control operators is complicated by the large number of transformations in use. Inspired by Section 8 of RFC 8949 [STD94], we use representations defined in [RFC4648] with the following names:
<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>.b64u</td>
<td>Base64URL, no padding</td>
<td>Section 5 of [RFC4648]</td>
</tr>
<tr>
<td>.b64u-sloppy</td>
<td>Base64URL, no padding, sloppy</td>
<td>Section 5 of [RFC4648]</td>
</tr>
<tr>
<td>.b64c</td>
<td>Base64 classic, padding</td>
<td>Section 4 of [RFC4648]</td>
</tr>
<tr>
<td>.b64c-sloppy</td>
<td>Base64 classic, padding, sloppy</td>
<td>Section 4 of [RFC4648]</td>
</tr>
<tr>
<td>.b32</td>
<td>Base32, no padding</td>
<td>Section 6 of [RFC4648]</td>
</tr>
<tr>
<td>.h32</td>
<td>Base32/hex alphabet, no padding</td>
<td>Section 7 of [RFC4648]</td>
</tr>
<tr>
<td>.hex</td>
<td>Base16 (hex), either case</td>
<td>Section 8 of [RFC4648]</td>
</tr>
<tr>
<td>.hex1c</td>
<td>Base16 (hex), lower case</td>
<td>Section 8 of [RFC4648]</td>
</tr>
<tr>
<td>.hexuc</td>
<td>Base16 (hex), upper case</td>
<td>Section 8 of [RFC4648]</td>
</tr>
<tr>
<td>.b45</td>
<td>Base45</td>
<td>[RFC9285]</td>
</tr>
</tbody>
</table>

Table 2: Control Operators for Text Conversion of byte strings

Note that this specification is somewhat opinionated here: It does not provide base64url, base32 or base32hex encoding with padding, or base64 classic without padding. Experience indicates that these combinations only ever occur in error, so the usability of CDDL is increased by not providing them in the first place. Also, adding "c" makes sure that any decision for classic base64 is actively taken.

The additional designation "sloppy" indicates that the text string is not validated for any additional bits being zero, in variance to what is specified in the paragraph behind table 1 in Section 4 of [RFC4648]. Note that the present specification is opinionated again in not specifying a sloppy variant of base32 or base32/hex, as no legacy use of sloppy base32(/hex) was known at the time of writing. Base45 is known to be suboptimal for use in environments with limited data transparency (such as URLs), but is included because of its
close relationship to QR codes and its wide use in health informatics
(note that base45 is strongly specified not to allow sloppy forms of
encoding).

2.2. Numbers

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>.decimal</td>
<td>Decimal Integer</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 3: Control Operator for Text
Conversion of Integers

The control operator .decimal allows the modeling of text strings
that carry numeric information in decimal form, such as in the
uint64/int64 formats of YANG-JSON [RFC7951].

yang-json-sid = text .decimal (0..9223372036854775807)

Again, the specification is opinionated by only providing integer
numbers without leading zeros, i.e., the decimal numbers match the
regular expression 0|-?[1-9][0-9]* (of course, further restricted by
the control type). See the next section for more flexibility, and
for octal, hexadecimal, or binary conversions.

2.3. Printf-style Formatting

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>.printf</td>
<td>Printf-formatting of data item(s)</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 4: Control Operator for Printf-formatting of data
item(s)

The control operator .printf allows the modeling of text strings
that carry various formatted information, as long as the format can be
represented in Printf-style formatting strings as they are used in
the C language (see Section 7.21.6.1 of [C]).
The controller (right-hand side) of the .printf control is an array of one Printf-style format string and zero or more data items that fit the individual conversion specifications in the format string. The construct matches a text string representing the textual output of an equivalent C-language printf function call that is given the format string and the data items following it in the array.

From the printf specification in the C language, length modifiers (paragraph 7) are not used and MUST NOT be included in the format string. The ‘s’ conversion specifier (paragraph 8) is used to interpolate a text string in UTF-8 form. The ‘c’ conversion specifier (paragraph 8) represents a single Unicode scalar value as a UTF-8 character. The ‘p’ and ‘n’ conversion specifiers (paragraph 8) are not used and MUST NOT be included in the format string.

In the following example, my_alg_19 matches the text string "0x0013":

```plaintext
my_alg_19 = hexlabel<19>
hexlabel<K> = text .printf (["0x%04x", K])
```

The data items in the controller array do not need to be literals, as for example in:

```plaintext
any_alg = hexlabel<1..20>
hexlabel<K> = text .printf (["0x%04x", K])
```

Here, any_alg matches the text strings "0x0013" or "0x0001" but not "0x1234".

2.4. JSON Values

Some applications store complete JSON texts into text strings, the JSON value for which can easily be defined in CDDL. This is supported by a control operator similar to .cbor in Section 3.8.4 of [RFC8610].

```
+----------+----------+------------+
| name     | meaning  | reference  |
|----------+----------+------------|
| .json    | JSON     | [STD90]    |
+----------+----------+------------+
```

Table 5: Control Operator for Text Conversion of JSON values

```plaintext
embedded-claims = text .json claims
claims = (iss: text, exp: text)
```
Note that a `.jsonseq` is not provided for [RFC7464], as no use case for inclusion in CDDL is known yet.

There is no way to constrain the use of blank space in data items to be validated; variants (e.g., not providing for any blank space) could be defined.

3. Text Processing

3.1. Join

Often, text strings need to be constructed out of parts that can best be modeled as an array.

```
+----------+------------------------------------------------+-----------+
| name     | meaning                                         | reference |
+----------+------------------------------------------------+-----------+
| .join    | concatenate elements of an array                | ---       |
+----------+------------------------------------------------+-----------+
```

Table 6: Control Operator for Text Generation from Arrays

In general, this control operator is hard to validate as it would require full parser functionality. It is therefore recommended to only use it in simple cases, and leave full parsing to ABNF (see Section 3 of [RFC9165]) or similar.

```
legacy-ip-address = text .join [bytetext, ".", bytetext, ".", bytetext, ".", bytetext]
bytetext = text .decimal byte
byte = 0..255
```

4. IANA Considerations

// RFC Editor: please replace RFC-XXXX with the RFC number of this // RFC and remove this note.

This document requests IANA to register the contents of Table 7 into the registry "CDDL Control Operators" of [IANA.cddl]:
5. Implementation Status

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

In the CDDL tool described in Appendix F of [RFC8610], the control operators defined in the present revision of this specification are implemented as of version 0.10.4.

6. Security considerations

The security considerations of [RFC8610] apply.
7. References

7.1. Normative References

[BCP14]  Best Current Practice 14, <https://www.rfc-editor.org/info/bcp14>. At the time of writing, this BCP comprises the following:


[STD90] Internet Standard 90, <https://www.rfc-editor.org/info/std90>. At the time of writing, this STD comprises the following:


[STD94] Internet Standard 94, <https://www.rfc-editor.org/info/std94>. At the time of writing, this STD comprises the following:


7.2. Informative References


Acknowledgements

Henk Birkholz suggested the need for many of the control operators defined here.
Author’s Address

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org
Abstract

CBOR (STD 94, RFC 8949) defines "Deterministically Encoded CBOR" in its Section 4.2, providing some flexibility for application specific decisions. To facilitate Deterministic Encoding to be offered as a selectable feature of generic encoders, the present document defines a CBOR Common Deterministic Encoding (CDE) Profile that can be shared by a large set of applications with potentially diverging detailed requirements.

This document also introduces the concept of Application Profiles, which are layered on top of the CBOR CDE Profile and can address more application specific requirements. Application Profiles are defined in separate documents.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on 4 September 2024.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction ........................................... 3
   1.1. Conventions and Definitions ........................... 3
2. CBOR Common Deterministic Encoding Profile (CDE) ............ 3
3. Application Profiles ..................................... 5
4. CDDL support ........................................... 6
5. Security Considerations .................................. 7
6. IANA Considerations ..................................... 7
7. References .............................................. 8
   7.1. Normative References ................................. 8
   7.2. Informative References ............................... 9
Appendix A. Implementers’ Checklists .......................... 9
   A.1. Preferred Serialization ............................... 10
      A.1.1. Preferred Serialization Encoders .................. 10
      A.1.2. Preferred Serialization Decoders .................. 11
   A.2. CDE ................................................. 11
      A.2.1. CDE Encoders ...................................... 11
      A.2.2. CDE Decoders ...................................... 12
Acknowledgments ............................................. 12
Contributors ............................................... 12
Author’s Address .......................................... 12
1. Introduction

CBOR (STD 94, RFC 8949) defines "Deterministically Encoded CBOR" in its Section 4.2, providing some flexibility for application specific decisions. To facilitate Deterministic Encoding to be offered as a selectable feature of generic encoders, the present document defines a CBOR Common Deterministic Encoding (CDE) Profile that can be shared by a large set of applications with potentially diverging detailed requirements.

This document also introduces the concept of Application Profiles, which are layered on top of the CBOR CDE Profile and can address more application specific requirements. Application Profiles are defined in separate documents. [I-D.mcnally-deterministic-cbor] is an example for such a document.

1.1. Conventions and Definitions

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.

2. CBOR Common Deterministic Encoding Profile (CDE)

This specification defines the _CBOR Common Deterministic Encoding Profile_ (CDE) based on the _Core Deterministic Encoding Requirements_ defined for CBOR in Section 4.2.1 of RFC 8949 [STD94].

In many cases, CBOR provides more than one way to encode a data item, but also provides a recommendation for a _Preferred Serialization_. The _CoRE Deterministic Encoding Requirements_ generally pick the preferred serializations as mandatory; they also pick additional choices such as definite-length encoding. Finally, it defines a map ordering based on lexicographic ordering of the (deterministically) encoded map keys.

Note that this specific set of requirements is elective in principle, other variants of deterministic encoding can be defined (and have been, now being phased out slowly, as detailed in Section 4.2.3 of RFC 8949 [STD94]). In many applications of CBOR today, deterministic encoding is not used at all, as its restriction of choices can create some additional performance cost and code complexity.
[STD94]’s core requirements are designed to provide well-understood and easy-to-implement rules while maximizing coverage, i.e., the subset of CBOR data items that are fully specified by these rules, and also placing minimal burden on implementations.

Section 4.2.2 of RFC 8949 [STD94] picks up on the interaction of extensibility (CBOR tags) and deterministic encoding. CBOR itself uses some tags to increase the range of its basic generic data types, e.g., tags 2/3 extend the range of basic major types 0/1 in a seamless way. Section 4.2.2 of RFC 8949 [STD94] recommends handling this transition the same way as with the transition between different integer representation lengths in the basic generic data model, i.e., by mandating the preferred serialization for all integers (Section 3.4.3 of RFC 8949 [STD94]).

1. The CBOR Common Deterministic Encoding Profile (CDE) turns this recommendation into a mandate: Integers that can be represented by basic major type 0 and 1 are encoded using the deterministic encoding defined for them, and integers outside this range are encoded using the preferred serialization (Section 3.4.3 of RFC 8949 [STD94]) of tag 2 and 3 (i.e., no leading zero bytes).

Most tags capture more specific application semantics and therefore may be harder to define a deterministic encoding for. While the deterministic encoding of their tag internals is often covered by the Core Deterministic Encoding Requirements, the mapping of diverging platform application data types on the tag contents may be hard to do in a deterministic way; see Section 3.2 of [I-D.bormann-cbor-det] for more explanation as well as examples. As the CDE would continually need to address additional issues raised by the registration of new tags, this specification recommends that new tag registrations address deterministic encoding in the context of this Profile.

A particularly difficult field to obtain deterministic encoding for is floating point numbers, partially because they themselves are often obtained from processes that are not entirely deterministic between platforms. See Section 3.2.2 of [I-D.bormann-cbor-det] for more details. Section 4.2.2 of RFC 8949 [STD94] presents a number of choices, which need to be made to obtain a CBOR Common Deterministic Encoding Profile (CDE). Specifically, CDE specifies (in the order of the bullet list at the end of Section 4.2.2 of RFC 8949 [STD94]):

2. Besides the mandated use of preferred serialization, there is no further specific action for the two different zero values, e.g., an encoder that is asked by an application to represent a negative floating point zero will generate 0xf98000.
3. There is no attempt to mix integers and floating point numbers, i.e., all floating point values are encoded as the preferred floating-point representation that accurately represents the value, independent of whether the floating point value is, mathematically, an integral value (choice 2 of the second bullet).

4. There is no special handling of NaN values, except that the preferred serialization rules also apply to NaNs with payloads, using the canonical encoding of NaNs as defined in [IEEE754]. Specifically, this means that shorter forms of encodings for a NaN are used when that can be achieved by only removing trailing zeros in the payload. Further clarifying [IEEE754], the CBOR encoding uses a leading bit of 1 to encode a quiet NaN; encoding of signaling NaN is NOT RECOMMENDED but is achieved by using a leading bit of 0.

Typically, most applications that employ NaNs in their storage and communication interfaces will only use the NaN with payload 0, which therefore deterministically encodes as 0xf97e00.

5. There is no special handling of subnormal values.

6. The CBOR Common Deterministic Encoding Profile does not presume equivalence of basic floating point values with floating point values using other representations (e.g., tag 4/5).

The main intent here is to preserve the basic generic data model, so Application Profiles can make their own decisions within that data model. E.g., an application profile can decide that it only ever allows a single NaN value that would encoded as 0xf97e00, so a CDE implementation focusing on this application profile would not need to provide processing for other NaN values. Basing the definition of both CDE and Application Profiles on the generic data model of CBOR also means that there is no effect on CDDL [RFC8610], except where the data description documents encoding decision for byte strings carrying embedded CBOR.

3. Application Profiles

While the CBOR Common Deterministic Encoding Profile (CDE) provides for commonality between different applications of CBOR, it is useful to further constrain the set of data items handled in a group of applications (_exclusions_) and to define further mappings (_reductions_) that help the applications in such a group get by with the exclusions.
For example, the dCBOR Application Profile specifies the use of Deterministic Encoding as defined in Section 4.2 of RFC 8949 [STD94] (see also [I-D.bormann-cbor-det] for more information) together with some application-level rules. See [I-D.mcnally-deterministic-cbor] for a definition of the dCBOR Application Profile that makes use of CDE.

In general, the application-level rules specified by an Application Profile are based on the shared CBOR Common Deterministic Encoding Profile; they do not "fork" CBOR in the sense of requiring distinct generic encoder/decoder implementations.

An Application Profile implementation produces well-formed, deterministically encoded CBOR according to [STD94], and existing generic CBOR decoders will therefore be able to decode it, including those that check for Deterministic Encoding. Similarly, generic CBOR encoders will be able to produce valid CBOR that can be processed by Application Profile implementations, if handed Application Profile conforming data model level information from an application.

Please note that the separation between standard CBOR processing and the processing required by the Application Profile is a conceptual one: Instead of employing generic encoders/decoders, both Application Profile processing and standard CBOR processing can be combined into a encoder/decoder specifically designed for the Application Profile.

An Application Profile is intended to be used in conjunction with an application, which typically will use a subset of the CBOR generic data model, which in turn influences which subset of the application profile is used. As a result, an Application Profile itself places no direct requirement on what minimum subset of CBOR is implemented. For instance, an application profile might define rules for the processing of floating point values, but there is no requirement that implementations of that Application Profile support floating point numbers (or any other kind of number, such as arbitrary precision integers or 64-bit negative integers) when they are used with applications that do not use them.

4. CDDL support

[RFC8610] defines control operators to indicate that the contents of a byte string carries a CBOR-encoded data item (.cbor) or a sequence of CBOR-encoded data items (.cborseq).

CDDL specifications may want to specify that the data items should be encoded in Common CBOR Deterministic Encoding. This specification adds two CDDL control operators that can be used for this.
The control operators .cde and .cdeseq are exactly like .cbor and .cborseq except that they also require the encoded data item(s) to be in Common CBOR Deterministic Encoding.

For example, a byte string of embedded CBOR that is to be encoded according to CDE can be formalized as:

leaf = #6.24(bytes .cde any)

More importantly, if the encoded data item also needs to have a specific structure, this can be expressed by the right hand side (instead of using the most general CDDL type any here).

(Note that the .cborseq control operator does not enable specifying different deterministic encoding requirements for the elements of the sequence. If a use case for such a feature becomes known, it could be added.)

Obviously, Application Profiles can define similar control operators that also embody the processing required by the Application Profile, and are encouraged to do so.

5. Security Considerations

The security considerations in Section 10 of RFC 8949 [STD94] apply. The use of deterministic encoding can mitigate issues arising out of the use of non-preferred serializations specially crafted by an attacker. However, this effect only accrues if the decoder actually checks that deterministic encoding was applied correctly. More generally, additional security properties of deterministic encoding can rely on this check being performed properly.

6. IANA Considerations

// RFC Editor: please replace RFCXXXX with the RFC number of this RFC
// and remove this note.

This document requests IANA to register the contents of Table 1 into the registry "CDDL Control Operators" of [IANA.cddl]:

Bormann                 Expires 4 September 2024                [Page 7]
### Table 1: New control operators to be registered

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cde</td>
<td>[RFCXXXX]</td>
</tr>
<tr>
<td>.cdeseq</td>
<td>[RFCXXXX]</td>
</tr>
</tbody>
</table>

7. References

7.1. Normative References

- [STD94] Internet Standard 94, <https://www.rfc-editor.org/info/std94>. At the time of writing, this STD comprises the following:

7.2. Informative References

[I-D.bormann-cbor-det]

[I-D.mcnally-deterministic-cbor]

Appendix A. Implementers’ Checklists

This appendix is informative. It provides brief checklists that implementers can use to check their implementations. It uses [RFC2119] language, specifically the keyword MUST, to highlight the specific items that implementers may want to check. It does not contain any normative mandates. This appendix is informative.

Notes:

* This is largely a restatement of parts of Section 4 of RFC 8949 [STD94]. The purpose of the restatement is to aid the work of implementers, not to redefine anything.

* Duplicate map keys are never valid in CBOR at all (see list item "Major type 5" in Section 3.1 of RFC 8949 [STD94]) no matter what sort of serialization is used. Of the various strategies listed in Section 5.6 of RFC 8949 [STD94], detecting duplicates and handling them as an error instead of passing invalid data to the application is the most robust one; achieving this level of robustness is a mark of quality of implementation.

* Preferred serialization and CDE only affect serialization. They do not place any requirements, exclusions, mappings or such on the data model level. Application profiles such as dCBOR are different as they can affect the data model by restricting some values and ranges.

* CBOR decoders in general are not required to check for preferred serialization or CDE and reject inputs that do not do not fulfill their requirements. However, in an environment that employs deterministic encoding, this negates many of its benefits. Decoder implementations that advertise "support" for preferred
serialization or CDE need to check the encoding and reject input that is not encoded to the encoding specification in use. Again, application profiles such as dCBOR may pose additional requirements, such as requiring rejection of non-conforming inputs.

If a generic decoder needs to be used that does not "support" CDE, a simple (but somewhat clumsy) way to check for proper CDE encoding is to re-encode the decoded data and check for bit-to-bit equality with the original input.

A.1. Preferred Serialization

In the following, the abbreviation "ai" will be used for the 5-bit additional information field in the first byte of an encoded CBOR data item, which follows the 3-bit field for the major type.

A.1.1. Preferred Serialization Encoders

1. Shortest-form encoding of the argument MUST be used for all major types. Major type 7 is used for floating-point and simple values; floating point values have its specific rules for how the shortest form is derived for the argument. The shortest form encoding for any argument that is not a floating point value is:

   * 0 to 23 and -1 to -24 MUST be encoded in the same byte as the major type.
   * 24 to 255 and -25 to -256 MUST be encoded only with an additional byte (ai = 0x18).
   * 256 to 65535 and -257 to -65536 MUST be encoded only with an additional two bytes (ai = 0x19).
   * 65536 to 4294967295 and -65537 to -4294967296 MUST be encoded only with an additional four bytes (ai = 0x1a).

2. If maps or arrays are emitted, they MUST use definite-length encoding (never indefinite-length).

3. If text or byte strings are emitted, they MUST use definite-length encoding (never indefinite-length).

4. If floating-point numbers are emitted, the following apply:

   * The length of the argument indicates half (binary16, ai = 0x19), single (binary32, ai = 0x1a) and double (binary64, ai = 0x1b) precision encoding. If multiple of these encodings
preserve the precision of the value to be encoded, only the shortest form of these MUST be emitted. That is, encoders MUST support half-precision and single-precision floating point. Positive and negative infinity and zero MUST be represented in half-precision floating point.

* NaNs, and thus NaN payloads MUST be supported.

As with all floating point numbers, NaNs with payloads MUST be reduced to the shortest of double, single or half precision that preserves the NaN payload. The reduction is performed by removing the rightmost \( N \) bits of the payload, where \( N \) is the difference in the number of bits in the significand (mantissa) between the original format and the reduced format. The reduction is performed only (preserves the value only) if all the rightmost bits removed are zero. (This will always reduce a double or single quiet NaN with a zero NaN payload to a half-precision quiet NaN.)

A.1.2. Preferred Serialization Decoders

1. Decoders MUST accept shortest-form encoded arguments.

2. If arrays or maps are supported, definite-length arrays or maps MUST be accepted.

3. If text or byte strings are supported, definite-length text or byte strings MUST be accepted.

4. If floating-point numbers are supported, the following apply:
   * Half-precision values MUST be accepted.
   * Double- and single-precision values SHOULD be accepted; leaving these out is only foreseen for decoders that need to work in exceptionally constrained environments.
   * If double-precision values are accepted, single-precision values MUST be accepted.
   * NaNs, and thus NaN payloads, MUST be accepted.

A.2. CDE

A.2.1. CDE Encoders

1. CDE encoders MUST only emit CBOR fulfilling the preferred serialization rules (Appendix A.1.1).
2. CDE encoders MUST sort maps by the CBOR representation of the map key. The sorting is byte-wise lexicographic order of the encoded map key data items.

A.2.2. CDE Decoders

1. CDE decoders MUST follow the rules for preferred serialization decoders (Appendix A.1.2).

Acknowledgments

An earlier version of this document was based on the work of Wolf McNally and Christopher Allen as documented in [I-D.mcnally-deterministic-cbor]; more recent revisions of that document now make use of the present document and the concept of Application Profile. We would like to explicitly acknowledge that this work has contributed greatly to shaping the concept of a CBOR Common Deterministic Encoding and Application Profiles on top of that.

Contributors

Laurence Lundblade
Security Theory LLC
Email: lgl@securitytheory.com

Laurence provided the text that became Appendix A.

Author’s Address

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org
Abstract

The Concise Binary Object Representation, CBOR (STD 94, RFC 8949), defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data.

This document specifies how to add application-oriented extensions to the diagnostic notation. It then defines two such extensions for text representations of epoch-based date/times and of IP addresses and prefixes (RFC 9164).

A few further additions close some gaps in usability. To facilitate tool interoperation, this document specifies a formal ABNF definition for extended diagnostic notation (EDN) that accommodates application-oriented literals.

About This Document

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

The latest revision of this draft can be found at https://cbor-wg.github.io/edn-literal/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-cbor-edn-literals/.

Discussion of this document takes place on the cbor Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cbor-wg/edn-literal.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on 19 November 2024.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction ................................. 3
   1.1. Terminology ................................ 4
   1.2. (Non-)Objectives of this Document ........ 5
2. Application-Oriented Extension Literals ........... 6
   2.1. The "dt" Extension ........................ 7
   2.2. The "ip" Extension ........................ 8
3. Stand-in Representations in Binary CBOR .......... 9
   3.1. Handling unknown application-extension identifiers ... 9
   3.2. Handling information deliberately elided from an EDN document .......... 10
4. IANA Considerations ................................ 12
   4.1. CBOR Diagnostic Notation Application-extension Identifiers Registry .......... 12
   4.2. Encoding Indicators ......................... 13
   4.3. Media Type ................................ 14
   4.4. Content-Format ............................. 15
   4.5. Stand-in Tags .............................. 16
5. Security considerations .......................... 16
6. References ..................................... 16
   6.1. Normative References ...................... 16
6.2. Informative References ........................................ 19
Appendix A. ABNF Definitions ........................................ 19
   A.1. Overall ABNF Definition for Extended Diagnostic
        Notation .................................................... 20
   A.2. ABNF Definitions for app-string Content .................... 24
      A.2.1. h: ABNF Definition of Hexadecimal representation of a
              byte string .............................................. 24
      A.2.2. b64: ABNF Definition of Base64 representation of a byte
              string ..................................................... 24
      A.2.3. dt: ABNF Definition of RFC 3339 Representation of a
              Date/Time ............................................... 25
      A.2.4. ip: ABNF Definition of Textual Representation of an IP
              Address ...................................................... 25
Appendix B. EDN and CDDL .............................................. 26
Acknowledgements ...................................................... 26
Author’s Address ...................................................... 26

1. Introduction

For the Concise Binary Object Representation, CBOR, Section 8 of RFC 8949 [STD94] in conjunction with Appendix G of [RFC8610] defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data. Diagnostic notation syntax is based on JSON, with extensions for representing CBOR constructs such as binary data and tags. (Standardizing this together with the actual interchange format does not serve to create another interchange format, but enables the use of a shared diagnostic notation in tools for and in documents about CBOR.)

This document specifies how to add application-oriented extensions to the diagnostic notation. It then defines two such extensions for text representations of epoch-based date/times and of IP addresses and prefixes [RFC9164].

A few further additions close some gaps in usability. To facilitate tool interoperation, this document specifies a formal ABNF definition for extended diagnostic notation (EDN) that accommodates application-oriented literals. (See Appendix A.1 for an overall ABNF grammar as well as the ABNF definitions in Appendix A.2 for grammars for both the byte string presentations predefined in [STD94] and the application-extensions).

In addition, this document finally registers a media type identifier and a content-format for CBOR diagnostic notation. This does not elevate its status as an interchange format, but recognizes that interaction between tools is often smoother if media types can be used.
Examples in RFCs often do not use media type identifiers, but special sourcecode type names that are allocated in  https://www.rfc-editor.org/materials/sourcecode-types.txt  (https://www.rfc-editor.org/materials/sourcecode-types.txt). At the time of writing, this resource lists four sourcecode type names that can be used in RFCs for including CBOR data items and CBOR-related languages:

* cbor (which is actually not useful, as CBOR is a binary format and cannot be used in textual examples in an RFC),

* cbor-diag (which is another name for EDN, as defined in the present document),

* cbor-pretty (which is a possibly annotated and pretty-printed hexdump of an encoded CBOR data item, along the lines of the grammar of Appendix A.2.1, as used for instance for some the examples in Appendix A.3 of [RFC9290]), and

* cddl (which is used for the Concise Data Definition Language, CDDL, see Section 1.1 below).

1.1. Terminology

Section 8 of RFC 8949 [STD94] defines the original CBOR diagnostic notation, and Appendix G of [RFC8610] supplies a number of extensions to the diagnostic notation that result in the Extended Diagnostic Notation (EDN). The diagnostic notation extensions include popular features such as embedded CBOR (encoded CBOR data items in byte strings) and comments. A simple diagnostic notation extension that enables representing CBOR sequences was added in Section 4.2 of [RFC8742]. As diagnostic notation is not used in the kind of interchange situations where backward compatibility would pose a significant obstacle, there is little point in not using these extensions.

Therefore, when we refer to "diagnostic notation", we mean to include the original notation from Section 8 of RFC 8949 [STD94] as well as the extensions from Appendix G of [RFC8610], Section 4.2 of [RFC8742], and the present document. However, we stick to the abbreviation "EDN" as it has become quite popular and is more sharply distinguishable from other meanings than "DN" would be.

In a similar vein, the term "ABNF" in this document refers to the language defined in [STD68] as extended in [RFC7405], where the "characters" of Section 2.3 of RFC 5234 [STD68] are Unicode scalar values.
The term "CDDL" (Concise Data Definition Language) refers to the data definition language defined in [RFC8610] and its registered extensions (such as those in [RFC9165]), as well as [I-D.ietf-cbor-update-8610-grammar]. Additional information about the relationship between the two languages EDN and CDDL is captured in the informative Appendix B.

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

1.2. (Non-)Objectives of this Document

Section 8 of RFC 8949 [STD94] states the objective of defining a human-readable diagnostic notation with CBOR. In particular, it states:

| All actual interchange always happens in the binary format. |

One important application of EDN is the notation of CBOR data for humans: in specifications, on whiteboards, and for entering test data. A number of features, such as comments in string literals, are mainly useful for people-to-people communication via EDN. Programs also often output EDN for diagnostic purposes, such as in error messages or to enable comparison (including generation of diffs via tools) with test data.

For comparison with test data, it is often useful if different implementations generate the same (or similar) output for the same CBOR data items. This is comparable to the objectives of deterministic serialization for CBOR data items themselves (Section 4.2 of RFC 8949 [STD94]). However, there are even more representation variants in EDN than in binary CBOR, and there is little point in specifically endorsing a single variant as "deterministic" when other variants may be more useful for human understanding, e.g., the << >> notation as opposed to \h''; an EDN generator may have quite a few options that control what presentation variant is most desirable for the application that it is being used for.

Because of this, a deterministic representation is not defined for EDN, and there is no expectation for "roundtripping" from EDN to CBOR and back, i.e., for an ability to convert EDN to binary CBOR and back to EDN while achieving exactly the same result as the original input EDN the original EDN possibly was created by humans or by a different EDN generator.
However, there is a certain expectation that EDN generators can be configured to some basic output format, which:

* looks like JSON where that is possible;
* inserts encoding indicators only where the binary form differs from preferred encoding;
* uses hexadecimal representation (h’’) for byte strings, not b64’’ or embedded CBOR (<<>>);
* does not generate elaborate blank space (newlines, indentation) for pretty-printing, but does use common blank spaces such as after , and :

Additional features such as ensuring deterministic map ordering (Section 4.2 of RFC 8949 [STD94]) on output, or even deviating from the basic configuration in some systematic way, can further assist in comparing test data. Information obtained from a CDDL model can help in choosing application-oriented literals or specific string representations such as embedded CBOR or b64’’ in the appropriate places.

2. Application-Oriented Extension Literals

This document extends the syntax used in diagnostic notation for byte string literals to also be available for application-oriented extensions.

As per Section 8 of RFC 8949 [STD94], the diagnostic notation can notate byte strings in a number of [RFC4648] base encodings, where the encoded text is enclosed in single quotes, prefixed by an identifier (»h« for base16, »b32« for base32, »h32« for base32hex, »b64« for base64 or base64url).

This syntax can be thought to establish a name space, with the names "h", "b32", "h32", and "b64" taken, but other names being unallocated. The present specification defines additional names for this namespace, which we call _application-extension identifiers_. For the quoted string, the same rules apply as for byte strings. In particular, the escaping rules that were adapted from JSON strings are applied equivalently for application-oriented extensions, e.g., within the quoted string \ stands for a single backslash and \ stands for a single quote.

An application-extension identifier is a name consisting of a lower-case ASCII letter (a-z) and zero or more additional ASCII characters that are either lower-case letters or digits (a-z0-9).
Application-extension identifiers are registered in a registry (Section 4.1).

Prefixing a single-quoted string, an application-extension identifier is used to build an application-oriented extension literal, which stands for a CBOR data item the value of which is derived from the text given in the single-quoted string using a procedure defined in the specification for an application-extension identifier.

An application-extension (such as dt) MAY also define the meaning of a variant of the application-extension identifier where each lower-case character is replaced by its upper-case counterpart (such as DT), for building an application-oriented extension literal using that all-uppercase variant as the prefix of a single-quoted string.

As a convention for such definitions, using the all-uppercase variant implies making use of a tag appropriate for this application-oriented extension (such as tag number 1 for DT).

Examples for application-oriented extensions to CBOR diagnostic notation can be found in the following sections.

2.1. The "dt" Extension

The application-extension identifier "dt" is used to notate a date/time literal that can be used as an Epoch-Based Date/Time as per Section 3.4.2 of RFC 8949 [STD94].

The text of the literal is a Standard Date/Time String as per Section 3.4.1 of RFC 8949 [STD94].

The value of the literal is a number representing the result of a conversion of the given Standard Date/Time String to an Epoch-Based Date/Time. If fractional seconds are given in the text (production time-secfrac in Figure 4), the value is a floating-point number; the value is an integer number otherwise. In the all-upper-case variant of the app-prefix, the value is enclosed in a tag number 1.

As an example, the CBOR diagnostic notation

dt’1969-07-21T02:56:16Z’,
dt’1969-07-21T02:56:16.5Z’,
DT’1969-07-21T02:56:16Z’

is equivalent to
-14159024,
-14159023.5,
1(-14159024)

See Appendix A.2.3 for an ABNF definition for the content of dt literals.

2.2. The "ip" Extension

The application-extension identifier "ip" is used to notate an IP address literal that can be used as an IP address as per Section 3 of [RFC9164].

The text of the literal is an IPv4address or IPv6address as per Section 3.2.2 of [RFC3986].

With the lower-case app-string ip, the value of the literal is a byte string representing the binary IP address. With the upper-case app-string IP, the literal is such a byte string tagged with tag number 54, if an IPv6address is used, or tag number 52, if an IPv4address is used.

As an additional case, the upper-case app-string IP’’ can be used with a prefix such as 2001:db8::/56 or 192.0.2.0/24, with the equivalent tag as its value. (Note that [RFC9164] representations of address prefixes need to implement the truncation of the address byte string as described in Section 4.2 of [RFC9164]; see example below.) For completeness, the lower-case variant ip’2001:db8::/56’ or ip’192.0.2.0/24’ stands for an unwrapped [56,h’20010db8’] or [24,h’c00002’]; however, in this case the information on whether an address is IPv4 or IPv6 often needs to come from the context.

Note that there is no direct representation of an address combined with a prefix length; this can be represented as 52([ip’192.0.2.42’,24]), if needed.

Examples: the CBOR diagnostic notation

ip’192.0.2.42’,
IP’192.0.2.42’,
IP’192.0.2.0/24’,
IP’2001:db8::42’,
IP’2001:db8::42’,
IP’2001:db8::/64’

is equivalent to
3. Stand-in Representations in Binary CBOR

In some cases, an EDN consumer cannot construct actual CBOR items that represent the CBOR data intended for eventual interchange. This document defines stand-in representation for two such cases:

* The EDN consumer does not know (or does not implement) an application-extension identifier used in the EDN document (Section 3.1) but wants to preserve the information for a later processor.

* The generator of some EDN intended for human consumption (such as in a specification document) may not want to include parts of the final data item, destructively replacing complete subtrees or possibly just parts of a lengthy string by _elisions_ (Section 3.2).

3.1. Handling unknown application-extension identifiers

When ingesting CBOR diagnostic notation, any application-oriented extension literals are usually decoded and transformed into the corresponding data item during ingestion. If an application-extension is not known or not implemented by the ingesting process, this is usually an error and processing has to stop.

However, in certain cases, it can be desirable to exceptionally carry an uninterpreted application-oriented extension literal in an ingested data item, allowing to postpone its decoding to a specific later stage of ingestion.

This specification defines a CBOR Tag for this purpose: The Diagnostic Notation Unresolved Application-Extension Tag, tag number CPA999 (Section 4.5). The content of this tag is an array of two text strings: The application-extension identifier, and the (escape-processed) content of the single-quoted string. For example, 

\[ \text{dt}'1969-07-21T02:56:16Z' \]

can be provisionally represented as /CPA/999(["dt", "1969-07-21T02:56:16Z"]).

See Appendix A.2.4 for an ABNF definition for the content of ip literals.
3.2. Handling information deliberately elided from an EDN document

When using EDN for exposition in a document or on a whiteboard, it is often useful to be able to leave out parts of an EDN document that are not of interest at that point of the exposition.

To facilitate this, this specification supports the use of an _ellipsis_ (notated as three or more dots in a row, as in ...) to indicate parts of an EDN document that have been elided (and therefore cannot be reconstructed).

Upon ingesting EDN as a representation of a CBOR data item for further processing, the occurrence of an ellipsis usually is an error and processing has to stop.

However, it is useful to be able to process EDN documents with ellipses in the automation scripts for the documents using them. This specification defines a CBOR Tag that can be used in the ingestion for this purpose: The Diagnostic Notation Ellipsis Tag, tag number CPA888 (Section 4.5). The content of this tag either is

1. null (indicating a data item entirely replaced by an ellipsis), or it is

2. an array, the elements of which are alternating between fragments of a string and the actual elisions, represented as ellipses carrying a null as content.

Elisions can stand in for entire subtrees, e.g. in:

```
[1, 2, ..., 3]
{
  "a": 1,
  "b": ...,
  ...: ...
}
```

A single ellipsis (or key/value pair of ellipses) can imply eliding multiple elements in an array (members in a map); if more detailed control is required, a data definition language such as CDDL can be
employed. (Note that the stand-in form defined here does not allow multiple key/value pairs with an ellipsis as a key: the CBOR data item would not be valid.)

Subtree elisions can be represented in a CBOR data item by using /CPA/888(null) as the stand-in:

```
[1, 2, 888(null), 3]
,  
  "a": 1,
  "b": 888(null),
  888(null): 888(null)
}
```

Elisions also can be used as part of a (text or byte) string:

```
{ "contract": "Herewith I buy" ... "gned: Alice & Bob",
  "signature": h'4711...0815',
}
```

The example "contract" uses string concatenation as per Appendix G.4 of [RFC8610], extending that by allowing ellipses; while the example "signature" uses special syntax that allows the use of ellipses between the bytes notated _inside_ h’’ literals.

String elisions can be represented in a CBOR data item by a stand-in that wraps an array of string fragments alternating with ellipsis indicators:

```
{ "contract": /CPA/888( ["Herewith I buy", 888(null),
  "gned: Alice & Bob"]),
  "signature": 888([h'4711’, 888(null), h’0815’]),
}
```

Note that the use of elisions is different from "commenting out" EDN text, e.g.

```
{ "contract": "Herewith I buy" /.../ "gned: Alice & Bob",
  "signature": h'4711/.../0815’,
  # ...: ...
}
```

The consumer of this EDN will ignore the comments and therefore will have no idea after ingestion that some information has been elided; validation steps may then simply fail instead of being informed about the elisions.
4. IANA Considerations

// RFC Editor: please replace RFC-XXXX with the RFC number of this
// RFC, [IANA.cbor-diagnostic-notation] with a reference to the new
// registry group, and remove this note.

4.1. CBOR Diagnostic Notation Application-extension Identifiers

Registry

IANA is requested to create an "Application-Extension Identifiers"
registry in a new "CBOR Diagnostic Notation" registry group
[IANA.cbor-diagnostic-notation], with the policy "expert review"
(Section 4.5 of RFC 8126 [BCP26]).

The experts are instructed to be frugal in the allocation of
application-extension identifiers that are suggestive of generally
applicable semantics, keeping them in reserve for application-
extensions that are likely to enjoy wide use and can make good use of
their conciseness. The expert is also instructed to direct the
registrant to provide a specification (Section 4.6 of RFC 8126
[BCP26]), but can make exceptions, for instance when a specification
is not available at the time of registration but is likely
forthcoming. If the expert becomes aware of application-extension
identifiers that are deployed and in use, they may also initiate a
registration on their own if they deem such a registration can avert
potential future collisions.

Each entry in the registry must include:

Application-Extension Identifier:
a lower case ASCII [STD80] string that starts with a letter and
can contain letters and digits after that ([a-z][a-z0-9]*)
No other entry in the registry can have the same application-
extension identifier.

Description:
a brief description

Change Controller:
(see Section 2.3 of RFC 8126 [BCP26])

Reference:
a reference document that provides a description of the
application-extension identifier

The initial content of the registry is shown in Table 1; all initial
entries have the Change Controller "IETF".
4.2. Encoding Indicators

IANA is requested to create an "Encoding Indicators" registry in the newly created "CBOR Diagnostic Notation" registry group [IANA.cbor-diagnostic-notation], with the policy "specification required" (Section 4.6 of RFC 8126 [BCP26]).

The experts are instructed to be frugal in the allocation of encoding indicators that are suggestive of generally applicable semantics, keeping them in reserve for encoding indicator registrations that are likely to enjoy wide use and can make good use of their conciseness. If the expert becomes aware of encoding indicators that are deployed and in use, they may also solicit a specification and initiate a registration on their own if they deem such a registration can avert potential future collisions.

Each entry in the registry must include:

Encoding Indicator:
an ASCII [STD80] string that starts with an underscore letter and can contain zero or more underscores, letters and digits after that (__[A-Za-z0-9]__). No other entry in the registry can have the same Encoding Indicator.

Description:
a brief description. This description may employ an abbreviation of the form ai=nn, where nn is the numeric value of the field _additional information_, the low-order 5 bits of the initial byte (see Section 3 of RFC 8949 [STD94]).

<table>
<thead>
<tr>
<th>Application-extension Identifier</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>Reserved</td>
<td>RFC8949</td>
</tr>
<tr>
<td>b32</td>
<td>Reserved</td>
<td>RFC8949</td>
</tr>
<tr>
<td>h32</td>
<td>Reserved</td>
<td>RFC8949</td>
</tr>
<tr>
<td>b64</td>
<td>Reserved</td>
<td>RFC8949</td>
</tr>
<tr>
<td>dt</td>
<td>Date/Time</td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>ip</td>
<td>IP Address/Prefix</td>
<td>RFC-XXXX</td>
</tr>
</tbody>
</table>

Table 1: Initial Content of Application-extension Identifier Registry
Change Controller:
(see Section 2.3 of RFC 8126 [BCP26])

Reference:
a reference document that provides a description of the application-extension identifier

The initial content of the registry is shown in Table 2; all initial entries have the Change Controller "IETF".

<table>
<thead>
<tr>
<th>Encoding Indicator</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Indefinite Length</td>
<td>RFC8949,</td>
</tr>
<tr>
<td></td>
<td>Encoding (ai=31)</td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>_i</td>
<td>ai=0 to ai=23</td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>_0</td>
<td>ai=24</td>
<td>RFC8949,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>_1</td>
<td>ai=25</td>
<td>RFC8949,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>_2</td>
<td>ai=26</td>
<td>RFC8949,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFC-XXXX</td>
</tr>
<tr>
<td>_3</td>
<td>ai=27</td>
<td>RFC8949,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFC-XXXX</td>
</tr>
</tbody>
</table>

Table 2: Initial Content of Encoding Indicator Registry

4.3. Media Type

IANA is requested to add the following Media-Type to the "Media Types" registry [IANA.media-types].

<table>
<thead>
<tr>
<th>Name</th>
<th>Template</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbor-diagnostic</td>
<td>application/cbor-diagnostic</td>
<td>RFC-XXXX, Section 4.3</td>
</tr>
</tbody>
</table>

Table 3: New Media Type application/cbor-diagnostic
Type name: application
Subtype name: cbor-diagnostic
Required parameters: N/A
Optional parameters: N/A
Encoding considerations: binary (UTF-8)
Security considerations: Section 5 of RFC XXXX
Interoperability considerations: none
Published specification: Section 4.3 of RFC XXXX
Applications that use this media type: Tools interchanging a human-readable form of CBOR
Fragment identifier considerations: The syntax and semantics of fragment identifiers is as specified for "application/cbor". (At publication of RFC XXXX, there is no fragment identification syntax defined for "application/cbor".)
Additional information:
  Deprecated alias names for this type: N/A
  Magic number(s): N/A
  File extension(s): .diag
  Macintosh file type code(s): N/A
Person & email address to contact for further information: CBOR WG mailing list (cbor@ietf.org), or IETF Applications and Real-Time Area (art@ietf.org)
Intended usage: LIMITED USE
Restrictions on usage: CBOR diagnostic notation represents CBOR data items, which are the format intended for actual interchange. The media type application/cbor-diagnostic is intended to be used within documents about CBOR data items, in diagnostics for human consumption, and in other representations of CBOR data items that are necessarily text-based such as in configuration files or other data edited by humans, often under source-code control.
Author/Change controller: IETF
Provisional registration: no

4.4. Content-Format

IANA is requested to register a Content-Format number in the "CoAP Content-Formats" sub-registry, within the "Constrained RESTful Environments (CoRE) Parameters" Registry [IANA.core-parameters], as follows:
Table 4: New Content-Format

TBD1 is to be assigned from the space 256..9999, according to the procedure "IETF Review or IESG Approval", preferably a number less than 1000.

4.5. Stand-in Tags

// RFC-Editor: This document uses the CPA (code point allocation) convention described in [I-D.bormann-cbor-draft-numbers]. For each usage of the term "CPA", please remove the prefix "CPA" from the indicated value and replace the residue with the value assigned by IANA; perform an analogous substitution for all other occurrences of the prefix "CPA" in the document. Finally, please remove this note.

In the "CBOR Tags" registry [IANA.cbor-tags], IANA is requested to assign the tags in Table 5 from the "specification required" space (suggested assignments: 888 and 999), with the present document as the specification reference.

Table 5: Values for Tags

5. Security considerations

The security considerations of [STD94] and [RFC8610] apply.

6. References

6.1. Normative References
At the time of writing, this BCP comprises the following:


[STD68] Internet Standard 68, <https://www.rfc-editor.org/info/std68>. At the time of writing, this STD comprises the following:


[STD80] Internet Standard 80, <https://www.rfc-editor.org/info/std80>. At the time of writing, this STD comprises the following:

[STD94] Internet Standard 94, <https://www.rfc-editor.org/info/std94>. At the time of writing, this STD comprises the following:


6.2. Informative References


[STD90] Internet Standard 90, <https://www.rfc-editor.org/info/std90>. At the time of writing, this STD comprises the following:


Appendix A. ABNF Definitions

This appendix is normative.

It collects grammars in ABNF form ([STD68] as extended in [RFC7405]) that serve to define the syntax of EDN and some application-oriented literals.
Implementation note: The ABNF definitions in this appendix are intended to be useful in a Parsing Expression Grammar (PEG) parser interpretation (see Appendix A of [RFC8610] for an introduction into PEG).

A.1. Overall ABNF Definition for Extended Diagnostic Notation

This appendix provides an overall ABNF definition for the syntax of CBOR extended diagnostic notation.

To complete the parsing of an app-string with prefix, say, p, the processed sqstr inside it is further parsed using the ABNF definition specified for the production app-string-p in Appendix A.2.

For simplicity, the internal parsing for the built-in EDN prefixes is specified in the same way. ABNF definitions for h’’ and b64’’ are provided in Appendix A.2.1 and Appendix A.2.2. However, the prefixes b32’’ and h32’’ are not in wide use and an ABNF definition in this document could therefore not be based on implementation experience.

```plaintext
seq         = S [item S *(""," S item S) OC] S
one-item    = S item S
item        = map / array / tagged
             / number / simple
             / string / streamstring

stringl1    = (tstr / bstr) spec
stringl     = stringl1 / ellipsis
ellipsis    = 3*"." ; "..." or more dots
string      = stringl *(S stringl)

number      = (basenumber / decnumber / infin) spec
sign        = "+" / "-"

decnumber   = [sign] (1*DIGIT ["." *DIGIT] / "." 1*DIGIT)
              ["e" [sign] 1*DIGIT]

basenumber  = [sign] "0" ("x" 1*HEXDIG
              ["." *HEXDIG] "p" [sign] 1*DIGIT]
              / "x" "." 1*HEXDIG "p" [sign] 1*DIGIT
              / "o" 1*ODIGIT
              / "b" 1*BDIGIT)

infin       = %s"Infinity"
             / %s"-Infinity"
             / %s"NaN"

simple      = %s"false"
             / %s"true"
             / %s"null"
             / %s"undefined"
             / %s"simple( " S item S ")"
```
uint  = "0" / DIGIT1 *DIGIT
tagged = uint spec "(" S item S ")"

app-prefix = lcalpha *lcalnum ; including h and b64
            / ucalpha *ucalnum ; tagged variant, if defined
app-string = app-prefix sqstr
sqstr   = "/" *single-quoted "/
back slash = app-string / sqstr / embedded
            ; app-string could be any type
tstr    = DQUOTE *double-quoted DQUOTE
embedded = "<<" seq ">>"

array   = "[" spec S [item S *("," S item S) OC] "]"
map     = "{" spec S [kp S *("," S kp S) OC] "}"
kp      = item S ":" S item

; We allow %x09 HT in prose, but not in strings
blank   = %x09 / %x0A / %x0D / %x20
non-slash = blank / %x21-2e / %x30-D7FF / %xE000-10FFFF
non-lf  = %x09 / %x0D / %x20-D7FF / %xE000-10FFFF
S       = *blank *(comment *blank)
comment = "/" *non-slash "/
          / ";" *non-lf %x0A

; optional trailing comma (ignored)
OC      = [",", S]

; check semantically that strings are either all text or all bytes
; note that there must be at least one string to distinguish
streamstring = "(_" S string S *("(_" S string S) OC ")"
spec      = ["_" *wordchar]

double-quoted = unescaped
               / "\" DQUOTE
               / "\" escapable

single-quoted = unescaped
               / DQUOTE
               / "\" "\" "\"
               / "\" escapable

escapable   = %s"b" ; BS backspace U+0008
              / %s"f" ; FF form feed U+000C
              / %s"n" ; LF line feed U+000A
              / %s"r" ; CR carriage return U+000D
              / %s"t" ; HT horizontal tab U+0009
              / "\" ; / slash (solidus) U+002F (JSON!)
Figure 1

While an ABNF grammar defines the set of character strings that are considered to be valid EDN by this ABNF, the mapping of these character strings into the generic data model of CBOR is not always obvious.

The following additional items should help in the interpretation:
* decnumber stands for an integer in the usual decimal notation, unless at least one of the optional parts starting with "." and "e" are present, in which case it stands for a floating point value in the usual decimal notation. Note that the grammar now allows 3. for 3.0 and .3 for 0.3 (also for hexadecimal floating point below); implementers are advised that some platform numeric parsers accept only a subset of the floating point syntax in this document and may require some preprocessing to use here.

* basenumber stands for an integer in the usual base 16/hexadecimal ("0x"), base 8/octal ("0o"), or base 2/binary ("0b") notation, unless the optional part containing a "p" is present, in which case it stands for a floating point number in the usual hexadecimal notation (which uses a mantissa in hexadecimal and an exponent in decimal notation, see Section 5.12.3 of [IEEE754], Section 6.4.4.2 of [C], or Section 5.13.4 of [Cplusplus]; floating-suffix/floating-point-suffix from the latter two is not used here).

* spec stands for an encoding indicator.

(In the following, an abbreviation of the form ai=nn gives nn as the numeric value of the field _additional information_, the low-order 5 bits of the initial byte: see Section 3 of RFC 8949 [STD94].)

As per Section 8.1 of RFC 8949 [STD94]:

- an underscore _ on its own stands for indefinite length encoding (ai=31, only available behind the opening brace/bracket for map and array: strings have a special syntax streamstring for indefinite length encoding except for the special cases "_" and "'_", and
- _0 to _3 stand for ai=24 to ai=27, respectively.

Surprisingly, Section 8.1 of RFC 8949 [STD94] does not address ai=0 to ai=23 the assumption seems to be that preferred serialization (Section 4.1 of RFC 8949 [STD94]) will be used when converting CBOR diagnostic notation to an encoded CBOR data item, so leaving out the encoding indicator for a data item with a preferred serialization will implicitly use ai=0 to ai=23 if that is possible. The present specification allows to make this explicit:

- _i ("immediate") stands for encoding with ai=0 to ai=23.
While no pressing use for further values for encoding indicators comes to mind, this is an extension point for EDN; Section 4.2 defines a registry for additional values.

* string and the rules preceding it in the same block realize both the representation of strings that are split up into multiple chunks (Appendix G.4 of [RFC8610]) and the use of ellipses to represent elisions (Section 3.2). The semantic processing of these rules is relatively complex:

- A single ... is a general ellipsis, which can stand for any data item.

- An ellipsis can be surrounded (on one or both sides) by string chunks, the result is a CBOR tag number CPA888 that contains an array with joined together spans of such chunks plus the ellipses represented by 888(null).

- A simple sequence of string chunks is simply joined together. In both cases of joining strings, the rules of Appendix G.4 of [RFC8610] need to be followed; in particular, if a text string results from the joining operation, that result needs to be valid UTF-8.

- Some of the strings may be app-strings. If the type of the app-string is an actual string, joining of chunked strings occurs as with directly notated strings; otherwise the occurrence of more than one app-string or an app-string together with a directly notated string cannot be processed.

A.2. ABNF Definitions for app-string Content

This appendix provides ABNF definitions for application-oriented extension literals defined in [STD94] and in this specification. These grammars describe the _decoded_ content of the sqstr components that combine with the application-extension identifiers to form application-oriented extension literals. Each of these may make use of rules defined in Figure 1.

A.2.1. h: ABNF Definition of Hexadecimal representation of a byte string

The syntax of the content of byte strings represented in hex, such as h’, h’0815’, or h’/head/ 63/ /contents/ 66 6f 6f’ (another representation of << "foo" >>), is described by the ABNF in Figure 2. This syntax accommodates both lower case and upper case hex digits, as well as blank space (including comments) around each hex digit.
The syntax of the content of byte strings represented in base64 is described by the ABNF in Figure 2.

This syntax allows both the classic (Section 4 of [RFC4648]) and the URL-safe (Section 5 of [RFC4648]) alphabet to be used. It accommodates, but does not require base64 padding. Note that inclusion of classic base64 makes it impossible to have in-line comments in b64, as "/" is valid base64-classic.

A.2.3. dt: ABNF Definition of RFC 3339 Representation of a Date/Time

The syntax of the content of dt literals can be described by the ABNF for date-time from [RFC3339] as summarized in Section 3 of [RFC9165]:

Figure 3: ABNF definition of Base64 Representation of a Byte String

A.2.4. app-string-h: ABNF Definition of Hexadecimal Representation of a Byte String

Figure 2: ABNF Definition of Hexadecimal Representation of a Byte String
app-string-dt   = date-time

date-fullyear   = 4DIGIT
date-month      = 2DIGIT  ; 01-12
date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                 ; month/year
time-hour       = 2DIGIT  ; 00-23
time-minute     = 2DIGIT  ; 00-59
time-second     = 2DIGIT  ; 00-58, 00-59, 00-60 based on leap sec
                 ; rules
time-secfrac    = "." 1*DIGIT
time-numoffset  = ("+" / "-") time-hour ":" time-minute
time-offset     = ":Z" / time-numoffset

partial-time    = time-hour ":" time-minute ":" time-second
                 [time-secfrac]
full-date       = date-fullyear "-" date-month "-" date-mday
full-time       = partial-time time-offset
date-time       = full-date "T" full-time
DIGIT           = %x30-39 ; 0-9

Figure 4: ABNF Definition of RFC3339 Representation of a Date/Time

A.2.4. ip: ABNF Definition of Textual Representation of an IP Address

The syntax of the content of ip literals can be described by the ABNF for
IPv4address and IPv6address in Section 3.2.2 of [RFC3986], as
included in slightly updated form in Figure 5.
app-string-ip = IPaddress ["/" uint]

IPaddress = IPv4address / IPv6address

; ABNF from RFC 3986, re-arranged for PEG compatibility:
IPv6address = 6( h16 "." ) ls32
   / "::" 5( h16 "." ) ls32
   / [ h16 ] "::" 4( h16 "." ) ls32
   / [ h16 *1( "" h16 ) ] "::" 3( h16 "." ) ls32
   / [ h16 *2( "" h16 ) ] "::" 2( h16 "." ) ls32
   / [ h16 *3( "" h16 ) ] "::" h16 "::" 1ls32
   / [ h16 *4( "" h16 ) ] "::" ls32
   / [ h16 *5( "" h16 ) ] "::" h16
   / [ h16 *6( "" h16 ) ] "::"

h16 = 1*4HEXDIG
ls32 = ( h16 "." h16 ) / IPv4address
IPv4address = dec-octet "." dec-octet "." dec-octet "." dec-octet
dec-octet = "25" %x30-35 ; 250-255
   / "2" %x30-34 DIGIT ; 200-249
   / "1" 2DIGIT ; 100-199
   / %x31-39 DIGIT ; 10-99
   / DIGIT ; 0-9

HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
DIGIT = %x30-39 ; 0-9
DIGIT1 = %x31-39 ; 1-9
uint = "0" / DIGIT1 *DIGIT

Figure 5: ABNF Definition of Textual Representation of an IP Address

Appendix B. EDN and CDDL

This appendix is informative.

EDN was designed as a language to provide a human-readable representation of an instance, i.e., a single CBOR data item or CBOR sequence. CDDL was designed as a language to describe an (often large) set of such instances (which itself constitutes a language), in the form of a _data definition_ or _grammar_ (or sometimes called _schema_).

The two languages share some similarities, not the least because they have mutually inspired each other. But they have very different roots:
* EDN syntax is an extension to JSON syntax [STD90]. (Any interoperable JSON text is also valid EDN.)

* CDDL syntax is inspired by ABNF’s syntax [STD68].

For engineers that are using both EDN and CDDL, it is easy to write "CDDLisms" or "EDNisms" into their drafts that are meant to be in the other language. (This is one more of the many motivations to always validate formal language instances with tools.)

Important differences include:

* Comment syntax. CDDL inherits ABNF’s semicolon-delimited end of line characters, while EDN finds nothing in JSON that could be inherited here. Inspired by JavaScript, EDN simplifies JavaScript’s copy of the original C comment syntax to be delimited by single slashes (where line ends are not of interest); it also adds end-of-line comments starting with #.

EDN:

```json
{ / alg / 1: -7 / ECDSA 256 / }
```

```json
{ 1:   # alg
   -7 # ECDSA 256 }
```

CDDL: ? 1 => int / tstr, ; algorithm identifier

* Syntax for tags. CDDL’s tag syntax is part of the system for referring to CBOR’s fundamentals (the major type 6, in this case) and (with [I-D.ietf-cbor-update-8610-grammar]) allows specifying the actual tag number separately, while EDN’s tag syntax is a simple decimal number and a pair of parentheses.

EDN:

```json
98([h'', # empty encoded protected header
   {}], # empty unprotected header
   ... # rest elided here
])
```

CDDL: COSE_Sign_Tagged = #6.98(COSE_Sign)
* Separator character. Like JSON, EDN requires commas as separators between array elements and map members (EDN also allows, but does not require, a trailing comma before the closing bracket/brace, enabling an easier to maintain "terminator" style of their use). CDDL’s comma separators in these contexts (CDDL groups) are entirely optional (and actually are terminators, which together with their optionality allows them to be used like separators as well, or even not at all).

* Embedded CBOR. EDN has a special syntax to describe the content of byte strings that are encoded CBOR data items. CDDL can specify these with a control operator, which looks very different.

EDN:

```plaintext
98([<< (/alg/ 1: -7 /ECDSA 256/>)>>, # == h’a10126’
    ... # rest elided here
])
```

CDDL: `serialized_map = bytes .cbor header_map`

Acknowledgements

The concept of application-oriented extensions to diagnostic notation, as well as the definition for the "dt" extension, were inspired by the CoRAL work by Klaus Hartke.

(TBD)

Author’s Address

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org
Abstract

The Concise Data Definition Language (CDDL), as defined in RFC 8610 and RFC 9165, provides an easy and unambiguous way to express structures for protocol messages and data formats that are represented in CBOR or JSON.

The present document updates RFC 8610 by addressing errata and making other small fixes for the ABNF grammar defined for CDDL there.

About This Document

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

The latest revision of this draft can be found at https://cbor-wg.github.io/update-8610-grammar/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-cbor-update-8610-grammar/.

Discussion of this document takes place on the CBOR Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cbor-wg/update-8610-grammar.

Status of This Memo

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

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

1. Introduction ................................................. 3
   1.1. Conventions and Definitions .......................... 3
2. Clarifications and Changes based on Errata Reports ........... 3
   2.1. Err6527 (text string literals) ...................... 3
   2.2. Err6543 (byte string literals) ....................... 5
       Change proposed by Errata Report 6543 ............... 5
       No change needed after addressing Err6527 (text string
       literals) (Section 2.1) ............................. 6
3. Small Enabling Grammar Changes .............................. 8
   3.1. Empty data models .................................. 8
   3.2. Non-literal Tag Numbers, Simple Values .............. 9
4. Security Considerations ...................................... 10
5. IANA Considerations .......................................... 10
6. References .................................................. 10
   6.1. Normative References ................................ 10
   6.2. Informative References ............................... 11
Appendix A. Updated Collected ABNF for CDDL ................... 12
Acknowledgments ................................................ 14
Author’s Address ................................................. 14
1. Introduction

The Concise Data Definition Language (CDDL), as defined in [RFC8610] and [RFC9165], provides an easy and unambiguous way to express structures for protocol messages and data formats that are represented in CBOR or JSON.

The present document updates [RFC8610] by addressing errata and making other small fixes for the ABNF grammar defined for CDDL there.

1.1. Conventions and Definitions

The Terminology from [RFC8610] applies. The grammar in [RFC8610] is based on ABNF, which is defined in [STD68] and [RFC7405].

2. Clarifications and Changes based on Errata Reports

A number of errata reports have been made around some details of text string and byte string literal syntax: [Err6527] and [Err6543]. These are being addressed in this section, updating details of the ABNF for these literal syntaxes. Also, [Err6526] needs to be applied (backslashes have been lost during RFC processing in some text explaining backslash escaping).

These changes are intended to mirror the way existing implementations have dealt with the errata. They also use the opportunity presented by the necessary cleanup of the grammar of string literals for a backward compatible addition to the syntax for hexadecimal escapes. The latter change is not automatically forward compatible (i.e., CDDL specifications that make use of this syntax do not necessarily work with existing implementations until these are updated, which this specification recommends).

2.1. Err6527 (text string literals)

The ABNF used in [RFC8610] for the content of text string literals is rather permissive:

; RFC 8610 ABNF:
text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-7E / %x80-10FFFD / SESC
SESC = "\" (%x20-7E / %x80-10FFFD)

Figure 1: Old ABNF for strings with permissive ABNF for SESC, but not allowing hex escapes
This allows almost any non-C0 character to be escaped by a backslash, but critically misses out on the \uXXXX and \uHHHH\uLLLLL forms that JSON allows to specify characters in hex (which should be applying here according to Bullet 6 of Section 3.1 of [RFC8610]). (Note that we import from JSON the unwieldy \uHHHH\uLLLLL syntax, which represents Unicode code points beyond U+FFFF by making them look like UTF-16 surrogate pairs; CDDL text strings are not using UTF-16 or surrogates.)

Both can be solved by updating the SESC production. We use the opportunity to add a popular form of directly specifying characters in strings using hexadecimal escape sequences of the form \u{hex}, where hex is the hexadecimal representation of the Unicode scalar value. The result is the new set of rules defining SESC in Figure 2:

; new rules collectively defining SESC:
SESC = "\" ( %x22 / " / / " / %x62 / %x66 / %x6E / %x72 / %x74 / %x75 hexchar )
hexchar = "{ (1="0" [ hexscalar ] / hexscalar )" } /
   non-surrogate / (high-surrogate "\" %x75 low-surrogate)
   non-surrogate = ((DIGIT / "A"/"B"/"C" / "E"/"F") 3HEXDIG) /
      ("D" %x30-37 2HEXDIG )
   high-surrogate = "D" ("8"/"9"/"A"/"B") 2HEXDIG
   low-surrogate = "D" ("C"/"D"/"E"/"F") 2HEXDIG
   hexscalar = "10" 4HEXDIG / HEXDIG1 4HEXDIG/
      non-surrogate / 1*3HEXDIG
HEXDIG1 = DIGIT1 / "A" / "B" / "C" / "D" / "E" / "F"

Figure 2: Updated string ABNF to allow hex escapes

(Notes: In ABNF, strings such as "A", "B" etc. are case-insensitive, as is intended here. We could have written %x62 as %s"b", but didn’t, in order to maximize ABNF tool compatibility.)

Now that SESC is more restrictively formulated, this also requires an update to the BCHAR production used in the ABNF syntax for byte string literals:

; RFC 8610 ABNF:
bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = "h" / "b64"

Figure 3: Old ABNF for BCHAR

With the SESC updated as above, \’ is no longer allowed in BCHAR; this now needs to be explicitly included.
Updating BCHAR also provides an opportunity to address [Err6278], which points to an inconsistency in treating U+007F (DEL) between SCHAR and BCHAR. As U+007F is not printable, including it in a byte string literal is as confusing as for a text string literal, and it should therefore be excluded from BCHAR as it is from SCHAR. The same reasoning also applies to the C1 control characters, so we actually exclude the entire range from U+007F to U+009F. The same reasoning then also applies to text in comments (PCHAR). For completeness, all these should also explicitly exclude the code points that have been set aside for UTF-16’s surrogates.

; new rules for BCHAR and SCHAR:
SCHAR = %x20-21 / %x23-5B / %x5D-7E / NONASCII / SESC
BCHAR = %x20-26 / %x28-5B / %x5D-7E / NONASCII / SESC / "\" / CRLF
PCHAR = %x20-7E / NONASCII
NONASCII = %xA0-D7FF / %xE000-10FFFD

Figure 4: Updated ABNF for BCHAR, SCHAR, and PCHAR

(Note that, apart from addressing the inconsistencies, there is no attempt to further exclude non-printable characters from the ABNF; doing this properly would draw in complexity from the ongoing evolution of the Unicode standard that is not needed here.)

2.2. Err6543 (byte string literals)

The ABNF used in [RFC8610] for the content of byte string literals lumps together byte strings notated as text with byte strings notated in base16 (hex) or base64 (but see also updated BCHAR production above):

; RFC 8610 ABNF:
bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF

Figure 5: Old ABNF for BCHAR

Change proposed by Errata Report 6543

Errata report 6543 proposes to handle the two cases in separate productions (where, with an updated SESC, BCHAR obviously needs to be updated as above):

; Err6543 proposal:
bytes = %x27 *BCHAR %x27
   / bsqual %x27 *QCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
QCHAR = DIGIT / ALPHA / "_" / "~" / "-" / "_" / "=" / WS
This potentially causes a subtle change, which is hidden in the WS production:

; RFC 8610 ABNF:
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-7E / %x80-10FFFD
CRLF = %x0A / %x0D.0A

Figure 7: ABNF definition of WS from RFC 8610

This allows any non-C0 character in a comment, so this fragment becomes possible:

```plaintext
foo = h'
  43424F52 ; 'CBOR'
  0A       ; LF, but don’t use CR!
,'
```

The current text is not unambiguously saying whether the three apostrophes need to be escaped with a \ or not, as in:

```plaintext
foo = h'
  43424F52 ; \'CBOR\'
  0A       ; LF, but don’t use CR!
,'
```

... which would be supported by the existing ABNF in [RFC8610].

No change needed after addressing Err6527 (text string literals) (Section 2.1)

This document takes the simpler approach of leaving the processing of the content of the byte string literal to a semantic step after processing the syntax of the bytes/BCHAR rules as updated by Figure 2 and Figure 4.

The rules in Figure 7 are therefore applied to the result of this processing where bsqual is given as h or b64.
Note that this approach also works well with the use of byte strings in Section 3 of [RFC9165]. It does require some care when copy-pasting into CDDL models from ABNF that contains single quotes (which may also hide as apostrophes in comments); these need to be escaped or possibly replaced by %x27.

Finally, our approach lends support to extending bsqual in CDDL similar to the way this is done for CBOR diagnostic notation in [I-D.ietf-cbor-edn-literals]. (Note that the processing of string literals now is quite similar between CDDL and EDN, except that CDDL has ";"-based end-of-line comments, while EDN has two comment syntaxes, in-line "/"-based and end-of-line "#"-based.)

The CDDL example in Figure 8 demonstrates various escaping techniques. Obviously in the literals for a and x, there is no need to escape the second character, an o, as \u{6f}; this is just for demonstration. Similarly, as shown in c and z there also is no need to escape the \ or , but escaping them may be convenient in order to limit the character repertoire of a CDDL file itself to ASCII [STD80].

\[
\text{start} = \{a, b, c, x, y, z\}
\]

; ",", DOMINO TILE VERTICAL-02-02, and
; ",", PLACE OF INTEREST SIGN, in a text string:
\[
a = \"D\u{6f}mino\'s \u{1F073} + \u{2318}\" \u{}
\]
\[
b = \"Domino\’s \uD83C\uDC73 + \u2318\" \\escape\JSON-like\n\]
\[
c = \"Domino\’s + \" \\unescape\n\]

; in a byte string given as text, the \ needs to be escaped:
\[
x = \'D\u{6f}mino\u{27}s \u{1F073} + \u{2318}\' \u{}
\]
\[
y = \'Domino\'s \uD83C\uDC73 + \u2318\' \\escape\JSON-like\n\]
\[
z = \'Domino\'s + \' \\escape\only\n\]

Figure 8: Example text and byte string literals with various escaping techniques

In this example, the rules a to c and x to z all produce strings with byte-wise identical content, where a to c are text strings, and x to z are byte strings. Figure 9 illustrates this by showing the output generated from the start rule in Figure 8, using pretty-printed hexadecimal.
3. Small Enabling Grammar Changes

The two subsections in this section specify two small changes to the grammar that are intended to enable certain kinds of specifications. These changes are backward compatible, i.e., CDDL files that comply to [RFC8610] continue to match the updated grammar, but not necessarily forward compatible, i.e., CDDL specifications that make use of these changes cannot necessarily be processed by existing [RFC8610] implementations.

3.1. Empty data models

[RFC8610] requires a CDDL file to have at least one rule.

; RFC 8610 ABNF:
cddl = S 1*(rule S)

This makes sense when the file has to stand alone, as a CDDL data model needs to have at least one rule to provide an entry point (start rule).

With CDDL modules [I-D.ietf-cbor-cddl-modules], CDDL files can also include directives, and these might be the source of all the rules that ultimately make up the module created by the file. Any other rule content in the file has to be available for directive processing, making the requirement for at least one rule cumbersome.

Therefore, we extend the grammar as in Figure 11 and make the existence of at least one rule a semantic constraint, to be fulfilled after processing of all directives.
Internet-Draft            CDDL grammar updates                  May 2024

; new top-level rule:
cddl = S *(rule S)

Figure 11: Updated ABNF for top-level rule cddl

3.2. Non-literal Tag Numbers, Simple Values

The existing ABNF syntax for expressing tags in CDDL is:

; extracted from RFC 8610 ABNF:
type2 =/ "#" "6" ["." uint] "(" S type S ")"

Figure 12: Old ABNF for tag syntax

This means tag numbers can only be given as literal numbers (uints).
Some specifications operate on ranges of tag numbers, e.g., [RFC9277]
has a range of tag numbers 1668546817 (0x63740101) to 1668612095
(0x6374FFFF) to tag specific content formats. This can currently not
be expressed in CDDL. Similar considerations apply to simple values
(#7.xx).

This update extends the syntax to:

; new rules collectively defining the tagged case:
type2 =/ "#" "6" ["." head-number] "(" S type S ")"
   / "#" "7" ["." head-number]
head-number = uint / ("<" type ">")

Figure 13: Updated ABNF for tag and simple value syntaxes

For #6, the head-number stands for the tag number. For #7, the head-
number stands for the simple value if it is in the ranges 0..23 or
32..255 (as per Section 3.3 of RFC 8949 [STD94] the simple values
24..31 are not used). For 24..31, the head-number stands for the
"additional information", e.g., #7.25 or #7.<25> is a float16, etc.
(All ranges mentioned here are inclusive.)

So the above range can be expressed in a CDDL fragment such as:

ct-tag<content> = #6.<ct-tag-number>(content)
ct-tag-number = 1668546817..1668612095
; or use 0x63740101..0x6374FFFF

Notes:
1. This syntax reuses the angle bracket syntax for generics; this
reuse is innocuous as a generic parameter/argument only ever
occurs after a rule name (id), while it occurs after . here.
(Whether there is potential for human confusion can be debated;
the above example deliberately uses generics as well.)

2. The updated ABNF grammar makes it a bit more explicit that the
number given after the optional dot is special, not giving the
CBOR "additional information" for tags and simple values as it is
with other uses of # in CDDL. (Adding this observation to
Section 2.2.3 of [RFC8610] is the subject of [Err6575]; it is
correctly noted in Section 3.6 of [RFC8610].) In hindsight,
maybe a different character than the dot should have been chosen
for this special case, however changing the grammar now would
have been too disruptive.

4. Security Considerations

The grammar fixes and updates in this document are not believed to
create additional security considerations. The security
considerations in Section 5 of [RFC8610] do apply, and specifically
the potential for confusion is increased in an environment that uses
a combination of CDDL tools some of which have been updated and some
of which have not been, in particular based on Section 2.

5. IANA Considerations

This document has no IANA actions.

6. References

6.1. Normative References

[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,

[STD68] Internet Standard 68,
<https://www.rfc-editor.org/info/std68>.
At the time of writing, this STD comprises the following:

Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
Specifications: ABNF", STD 68, RFC 5234,
DOI 10.17487/RFC5234, January 2008,
At the time of writing, this STD comprises the following:


6.2. Informative References


Appendix A. Updated Collected ABNF for CDDL

This appendix is normative.

It provides the full ABNF from [RFC8610] with the updates applied in the present document.

cddl = S *(rule S)
rule = typename [genericparm] S assignt S type
   / groupname [genericparm] S assigng S grpent

typename = id
groupname = id

assignt = "=" / "/="
assigng = "=" / "/="

genericparm = "<" S id S *("," S id S ) ">"
genericarg = "<" S type1 S *("," S type1 S ) ">"
type = type1 *(S "/" S type1)
type1 = type2 [S (rangeop / ctlop) S type2]
   ; space may be needed before the operator if type2 ends in a name

type2 = value
   / typename [genericarg]
   / "(" S type S ")"
   / "(" S group S ")"
   / "[" S group S "]"
   / "" S typename [genericarg]
   / "&" S "(" S group S ")"
   / "&" S groupname [genericarg]
   / ":" "6" ["." head-number] "(" S type S ")"
   / ":" "7" ["." head-number]
/ "#" DIGIT ["." uint] ; major/ai
/ "#" ; any
head-number = uint / ("<" type ">")
rangeop = "..." / "." .
ctlop = "." id
group = grpchoice *(S "/" S grpchoice)
grpchoice = *(grpent optcom)
grpent = [occur S] [memberkey S] type
   / [occur S] groupname [genericarg] ; preempted by above
   / [occur S] "(" S group S ")"
memberkey = type1 S ["^" S] "=>"
   / bareword S ":"
   / value S ":"

bareword = id

optcom = S ["," S]

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

uint = DIGIT1 *DIGIT
   / ":0" 1*HEXDIG
   / ":0b" 1*BINDIG
   / ":0"

value = number
   / text
   / bytes

int = ["-" ] uint

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

text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x25-3B / %x3D-7E / NONASCII / SESC
Internet-Draft            CDDL grammar updates                  May 2024

S = "\" ( %x22 / "/ "/ "\" / ; "/ "/ \\
%62 / %66 / %6E / %72 / %74 / ; \b \f \n \r \t
( %x75 hexchar ) ) ; \uXXXX

hexchar = "{" (1*"0" [ hexscalar ] / hexscalar) "}" /
         non-surrogate / (high-surrogate "\" %x75 low-surrogate)
non-surrogate = ((DIGIT / "A"/"B"/"C" / "E"/"F") 3HEXDIG) /
         ("D" %x30-37 2HEXDIG )
high-surrogate = "D" ("8"/"9"/"A"/"B") 2HEXDIG
low-surrogate = "D" ("C"/"D"/"E"/"F") 2HEXDIG
hexscalar = "10" 4HEXDIG / HEXDIG1 4HEXDIG
         / non-surrogate / 1*3HEXDIG

bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-7E / NONASCII / SESC / "\" / CRLF
bsqual = "h" / "b64"

id = EALPHA *("-" / ".") (EALPHA / DIGIT)
ALPHA = %x41-5A / %x61-7A
EALPHA = ALPHA / "@" / "_" / ";"
DIGIT = %x30-39
DIGIT1 = %x31-39
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
HEXDIG1 = DIGIT1 / "A" / "B" / "C" / "D" / "E" / "F"
BINDIG = %x30-31

S = *WS
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-7E / NONASCII
NONASCII = %xA0-D7FF / %xE000-10FFFF
CRLF = %x0A / %x0D.0A

Figure 14: ABNF for CDDL as updated

Acknowledgments

Many thanks go to the submitters of the errata reports addressed in
this document.  In one of the ensuing discussions, Doug Ewell
proposed to define an ABNF rule NONASCII, of which we have included
the essence.  Special thanks to the reviewers Marco Tiloca, Christian
Amsüss (shepherd review), and Orie Steele (AD review).

Author’s Address

Bormann                 Expires 18 November 2024               [Page 14]
Carsten Bormann
Universität Bremen Tzi
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org