Network Working Group
Internet-Draft
Updates: 8610, 8949 (if approved)
Intended status: Standards Track
Expires: 12 November 2026

C. Bormann
Universit辰t Bremen TZI

11 May 2026

CBOR Extended Diagnostic Notation (EDN)
draft-ietf-cbor-edn-literals-24

Abstract

This document formalizes and consolidates the definition of the Extended Diagnostic Notation (EDN) of the Concise Binary Object Representation (CBOR), addressing implementer experience.

Replacing EDN's previous informal descriptions, it updates RFC 8949, obsoleting its Section 8, and RFC 8610, obsoleting its Appendix G.

It also specifies registry-based extension points and uses them to support text representations such as of epoch-based dates/times and of IP addresses and prefixes.

// (This cref will be removed by the RFC editor:) The present -23 is
// intended as reference material during the 2026-04-29 CBOR interim,
// a complete specification that reacts to discussion on previous
// draft revisions and that can be used to confirm the results in a
// WGLC. Among other concerns, the discussion revealed that some
// additional editorial content was required; the attempt was to
// address this need without making technical changes.

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

Copyright Notice

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR) (RFC8949) [STD94] is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation. In addition to the binary interchange format, the original CBOR specification described a text-based "diagnostic notation" (Section 6 of [RFC7049], now Section 8 of RFC 8949 [STD94]), in order to be able to converse about CBOR data items without having to resort to binary data. Appendix G of [RFC8610] extended this into what is also known as Extended Diagnostic Notation (EDN).

Diagnostic notation syntax is based on JSON, with extensions for representing CBOR constructs such as binary data and tags.

Standardizing EDN in addition to the actual binary interchange format CBOR does not serve to create a competing interchange format, but enables the use of a shared diagnostic notation in tools for and in documents about CBOR. Still, between tools for CBOR development and diagnosis, document generation systems, continuous integration (CI) environments, configuration files, and user interfaces for viewing and editing for all these, EDN is often "interchanged" and therefore merits a specification that facilitates interoperability within this domain as well as reliable translation to and from CBOR. EDN is not designed or intended for general-purpose use in protocol elements exchanged between systems engaged in processes outside those listed here.

This document consolidates and formalizes the definition of EDN, providing a formal grammar (see Section 5.1 and Section 5.2), and incorporating small changes based on implementation experience. It updates RFC8949, obsoleting Section 8 of RFC 8949 [STD94], and [RFC8610], obsoleting Appendix G of [RFC8610]. It is intended to serve as a single reference target that can be used in specifications that use EDN.

It also specifies two registry-based extension points for the diagnostic notation: one for additional encoding indicators, and one for adding application-oriented literal forms. It uses these registries to add encoding indicators for a more complete coverage of encoding variation, and to add application-oriented literal forms that enhance EDN with text representations of epoch-based date/times, of IP addresses and prefixes [RFC9164], and of Concise Resource Identifiers (CRI [I-D.ietf-core-href]), as well as an application-oriented literal that represents cryptographic hash values computed from byte strings.

In addition, this document 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 Section 5.2.1, as used for instance for some of the examples in Appendix A.3 of [RFC9290]), and
>
> * cddl (which is used for the Concise Data Definition Language, CDDL, see Section 1.2 below).

Note that EDN is not meant to be the only text-based representation of CBOR data items. For instance, [YAML] [RFC9512] is able to represent most CBOR data items, possibly requiring use of YAML's extension points. YAML does not provide certain features that can be useful with tools and documents needing text-based representations of CBOR data items (such as embedded CBOR or encoding indicators), but it does provide a host of other features that EDN does not provide such as anchor/alias data sharing, at a cost of higher implementation and learning complexity.

1.1. Structure of This Document

Section 2 of this document has been built from Section 8 of RFC 8949 [STD94] and Appendix G of [RFC8610]. The latter provided a number of useful extensions to the initial diagnostic notation that was originally defined in Section 6 of [RFC7049]. Section 8 of RFC 8949 [STD94] and Appendix G of [RFC8610] have collectively been called "Extended Diagnostic Notation" (EDN), giving the present document its name.

After introductory material, Section 3 illustrates the concept of application-oriented extension literals by defining the "dt", "ip", "hash", and "cri" extensions. Section 4 defines mechanisms for dealing with unknown application-oriented literals and deliberately elided information. Section 5 gives the formal syntax of EDN in ABNF, with explanations for some features of and additions to this syntax, as an overall grammar (Section 5.1) and specific grammars for the content of app-string and byte-string literals (Section 5.2). This is followed by the conventional sections for IANA Considerations (6), Security considerations (7), and References (8.1, 8.2). An informational comparison of EDN with CDDL follows in Appendix A.

1.2. Terminology and Conventions

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; as at least some elements of the extended form are now near-universally used, the terms "diagnostic notation" and "EDN" have become synonyms in the context of CBOR.

Therefore, references to "_diagnostic notation_" generally 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, this document sticks 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. Brief snippets of grammar may be given in the text as I-Regexp regular expressions [RFC9485].

The term "CDDL" (Concise Data Definition Language) refers to the data definition language defined in [RFC8610] and its registered extensions (such as those documented in [RFC9165] and [RFC9682]). Additional information about the relationship between the two languages EDN and CDDL is captured in Appendix A.

Examples sometimes need to be quoted in the text, in particular in cases where the typewriter font used for example text cannot be distinguished in the plaintext rendition of this document. ASCII quotes, however, are already taken: true, "true", 'true', and ‘true‘ are all different literals in EDN and should not be confused. Therefore, a different quoting convention as in 損true束 or 損"true"束 is used for examples in the text where this is needed to remain unambiguous.

Superscript notation denotes exponentiation. For example, 2 to the power of 64+1 is notated: 2^(64+1). In the plain-text rendition of this specification, superscript notation is not available and exponentiation is therefore rendered by the surrogate notation seen here in the plain-text rendition.

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 [BCP14] (RFC2119) (RFC8174) when, and only when, they appear in all capitals, as shown here.

1.3. (Non-)Objectives of this Document

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

> All actual interchange always happens in the binary format.

1.3.1. For Humans

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 inside prefixed 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.

1.3.2. Determinism?

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.

1.3.3. Basic Output Format

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 Serialization (Section 4.1 of RFC 8949 [STD94]);
* 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 :.

EDN generators may provide configuration to consistently select either the unescaped (directly readable) or an escaped (ASCII equivalent) form of characters in string literals; the latter allows EDN to be used when the diagnostic value of fully escaped characters may be desired or in environments where non-ASCII characters may not enjoy full data transparency. Similar to JSON, EDN is designed to allow a simple tool to convert any EDN (including EDN with application extensions unknown to the tool) into fully escaped (printable ASCII and newlines only) form, as well as to inversely recover unescaped characters for all escapes where this is possible or for certain subsets of the characters (such as Unicode categories L, M, N, P, S, plus Zs or just ASCII space).

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. Overview over CBOR Extended Diagnostic Notation (EDN)

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

Note that diagnostic notation truly was designed as a diagnostic format; it originally was not meant to be parsed. Therefore, no formal definition (as in ABNF) was given in the original documents. Recognizing that formal grammars can aid interoperation of tools and usability of documents that employ EDN, Section 5 now provides ABNF definitions.

EDN is a true superset of JSON as it is defined in [STD90] in conjunction with [RFC7493] (that is, any interoperable [RFC7493] JSON text also is an EDN text), extending it both to cover the greater expressiveness of CBOR and to increase its usability.

EDN borrows the JSON syntax for numbers (integer and floating-point, Section 2.4), certain simple values (Section 2.8), UTF-8 [STD63] text strings, arrays, and maps (maps are called objects in JSON; the diagnostic notation extends JSON here by allowing any data item in the map key position).

As EDN is used for truly diagnostic purposes, its implementations MAY support generation and possibly ingestion of EDN for CBOR data items that are well-formed but not valid. It is RECOMMENDED that an implementation enables such usage only explicitly by configuration (such as an API or CLI flag). Validity of CBOR data items is discussed in Section 5.3 of RFC 8949 [STD94], with basic validity discussed in Section 5.3.1 of RFC 8949 [STD94], and tag validity discussed in Section 5.3.2 of RFC 8949 [STD94]. Tag validity is more likely a subject for individual application-oriented extensions, while the two cases of basic validity (for text strings and for maps) are addressed in Sections 2.5.7 and 2.6.3 under the heading of _validity_.

The rest of this section provides an overview over specific features of EDN, starting with certain common syntactical features and then going through kinds of CBOR data items roughly in the order of CBOR major types. Any additional detailed syntax discussion needed has been deferred to Section 5.1.

Additional information about implementation and use of EDN is continuously being collected by the community in [EDN-WIKI].

2.1. Application-Oriented Extension Literals

EDN provides _literals_ that represent CBOR data items textually. Many of the forms of literals provided are predefined by this document, but it also defines an extension point that enables defining additional _application-oriented extension literals_, or _extension literals_ for short.

Extension literals start with a _prefix_ that identifies the application-oriented extension, immediately followed by a sequence literal (Section 2.5.6) or a single-quoted or raw string literal (Section 2.5). The latter form uses its string literal as a shorthand form for a sequence literal representing a sequence with exactly that one text string data item.

> | This notation is generalized from Section 8 of RFC 8949 [STD94], which provides for notating byte strings in a number of [RFC4648] base encodings, where the encoded text is enclosed in single quotes, prefixed by a prefix (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 allows registering
> | additional names for this namespace, which it calls
> | _application-extension identifiers_.

More precisely, an _application-extension identifier_ is a registered name consisting of a lower-case ASCII letter ([a-z]) and zero or more additional ASCII characters that are either lower-case letters, digits, or hyphens ([a-z0-9-]). false, true, null, and undefined cannot be used as such identifiers and are reserved.

Application-extension identifiers are registered in the "Application-Extension Identifiers" registry (Section 6.1).

An application-extension (such as dt) MAY also define the meaning of one additional prefix derived from its application-extension identifier by replacing each lower-case character by its upper-case counterpart (such as DT). As a convention, using the all-uppercase variant implies making use of a CBOR tag appropriate for this application-oriented extension (such as tag number 1 for DT, where in contrast the prefix dt stands for the unwrapped tag content).

In summary, an application-extension identifier gives rise to one or two application-extension prefixes, one that is lexically identical to the identifier (i.e., all lowercase), and potentially another one that is an all-uppercase variation of it. In addition to specifying which of these two variations exhibits which specific semantics, the application extension specifies what input the extension takes.

When the prefix is used immediately in front of a single-quoted or a raw string, the input takes the form of a single text string CBOR data item. When used immediately in front of a sequence literal, the input is a CBOR sequence of elements of the sequence literal as input. The application extension can provide behavior that depends on the number of items supplied as input to it and their data types; it cannot distinguish between its prefix being used with a single-quoted string, a raw string, or a CBOR sequence composed of a single text string data item (as illustrated for instance in Tables 4, 5, and 6).

This specification defines a number of generally applicable application-oriented extensions (Section 3), both to motivate making these extensions generally available, and to illustrate the concept.

Of these, the application-oriented extensions h, b64, dt and ip are intended to be mandatory to implement. (As mentioned, for simplicity we use the term "application-oriented extensions" for the mechanism discussed in this section even if it is used to describe a part of base EDN.)

2.2. Comments

For presentation to humans, EDN text may benefit from comments. JSON famously does not provide for comments, and the original diagnostic notation in Section 6 of [RFC7049] inherited this property.

EDN now provides two comment syntaxes, which can be used where the syntax allows blank space (outside of constructs such as numbers, string literals, etc.):

* inline comments, delimited by slashes ("/") or by C-style "/*" and "*/":

  In a position that allows blank space, each of the following is considered blank space (and thus effectively a comment):

  - any text that starts with a slash followed by a character that is not a star or a slash, up to another slash, or

  - any text that starts with "/*" up to and including the next following "*/"

* end-of-line comments, delimited by "#" or "//" and an end of line (LINE FEED, U+000A):

  In a position that allows blank space, any text starting with "#" or "//" and ending with and including the end of the line is considered blank space (and thus effectively a comment).

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

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

This reduces to [1, 10584416, ["opsonize", 7, 105]].

Another example, combining the use of inline and end-of-line comments:

```
{
 /kty/ 1 : 4, # Symmetric
 /alg/ 3 : 5, # HMAC 256-256
  /k/ -1 : h'6684523ab17337f173500e5728c628547cb37df
              e68449c65f885d1b73b49eae1'
}
```

This reduces to {1: 4, 3: 5, -1: h'6684523AB17337F173500E5728C628547CB37DFE68449C65F885D1B73B49EAE1'}.

> Note that application-oriented extensions can define their own internal comment syntaxes for text inside strings, which may or may not mimic the overall comment syntax of EDN. The h'' syntax (Section 5.2.1), which the framework for application-oriented extensions was designed to include as an instance, provides an equivalent to the overall comment syntax inside its text strings. Similarly, b64'' (Section 5.2.2) provides a subset of that limited to "#" end-of-line comments (the slash character "/" is used in the alphabet in classic base64 encoding). None of the other application-oriented extensions supplied in this specification provides for such a kind of internal comment syntax.

2.2.1. Discussion

As a not quite backward compatible change, this specification restricts slash-delimited comments that were allowed in Appendix G.6 of [RFC8610] in two ways:

* Inline comments now longer can be empty: The construct "//" that was an empty comment in Appendix G.6 of [RFC8610] is now used instead to introduce an end-of-line comment. (Note that "//" still can be used in what is visually "within" a slash-delimited comment; its first slash actually ends the current comment and the second slash starts a new one.)

* EDN now enables the use of C-style inline comments: for instance, "/*foo/" was a complete comment in Appendix G.6 of [RFC8610] and now is the beginning of a C-style comment that goes on up to a "*/".

As an example, the introduction of C-style inline comments enables a comment explaining a COSE algorithm identifier, as in

```
4 /* HMAC 256/64 */
```

instead of the conventional, but often less familiar

```
4 / HMAC 256//64 /
```

2.3. Encoding Indicators

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

Encoding indicators are always optional: EDN is usually used to describe CBOR data items at the data model level. For some diagnostic purposes, it is useful to represent the choice of a serialization variation by including encoding indicators. Implementations of EDN generally do not need to provide this functionality in full; if they do, they can be called "diagnostic implementations". To be able to process EDN that contains encoding indicators, an EDN-consuming implementation MUST accept them (i.e., process or ignore the presence or absence of each encoding indicator). (Ignoring them could be compared to a generic CBOR decoder ignoring the presence of the serialization variants it encounters.) It is RECOMMENDED to by default provide a warning for each encoding indicator value that is encountered but not further processed.

When creating EDN as input for a diagnostic CBOR encoder in order to obtain specific encoding choices, encoding indicators may be placed manually or by the software generating the EDN. Where no encoding indicator is placed, a diagnostic CBOR encoder is expected to generate Preferred Serialization (Section 4.1 of RFC 8949 [STD94]) with definite length encoding only. Similarly, when using EDN as output for a diagnostic CBOR decoder, a basic diagnostic configuration of the tool is expected to provide encoding indicators only in places where the CBOR input did not use Preferred Serialization with definite length encoding (see also Section 1.3.3). Diagnostic implementations of EDN that process encoding indicators as discussed here are expected to document their diagnostic behavior and the processing options that can be selected.

2.3.1. Syntax, Semantics, Examples

Encoding indicators start with an underscore and comprise all immediately following characters that are alphanumeric or underscore. For example, _ or _3. Encoding indicators can be ignored by anyone not interested in this information.

Encoding indicators are placed immediately to the right of the data item or of a syntactic feature that can stand for the data item the encoding of which the encoding indicator is controlling. Table 1 provides examples for data items with encoding indicators used with various kinds of data items.

mt	examples
0	1_1, 0x4711_3
1	-1_1
2	'A'_1
3	"A"_1
4	[_1 "bar"]
5	{_1 "bar": 1}
6	1_1(4711)
7	1.5_2, 0x4711p+03_3

Table 1: Examples of Encoding Indicators for Different Data Items (mt = major type)

(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]. This field is used in encoding the "argument", i.e., the value, tag, or length; ai=0 to ai=23 mean that the value of the ai field immediately _is_ the argument, ai=24 to ai=27 mean that the argument is carried in 2^(ai-24) (1, 2, 4, or 8) additional bytes, and ai=31 means that indefinite length encoding is used.)

An underscore followed by a decimal digit n indicates that the preceding item (or, for arrays and maps, the item starting with the preceding bracket or brace) was or is to be encoded with an additional information value of ai=24+n. For example, 1.5_1 is a half-precision floating-point number (2^1 = 2 additional bytes or 16 bits), while 1.5_3 is encoded as double precision (2^3 = 8 additional bytes or 64 bits). For a tool consuming EDN in a diagnostic mode, encountering an encoding indicator that does not provide enough space to correctly encode the unchanged data item given is an error; there is no truncation or rounding that would change the data item encoded.

> | Truncation or rounding semantics imply performing changes at the data model level, which is outside the scope of encoding indicators. Such operations can be provided by application extensions.

The encoding indicator _ (an underscore on its own) is used to indicate indefinite length encoding. Indefinite length encoding uses ai=31, which could have been indicated by _7, which is therefore not used and marked as reserved (as are _4, _5, and _6, which would stand for ai=28 to ai=30, values currently not in use in CBOR; these encoding indicators will be available if and when CBOR is extended to make use of them).

Note that the encoding indicator _ is only available behind the opening brace/bracket for map and array (Section 2.6.2): strings have a special syntax streamstring for indefinite length encoding except for the special cases ''_ and ""_ (Section 2.5.4).

The encoding indicators _0 to _3 can be used to indicate ai=24 to ai=27, respectively; they therefore stand for 1, 2, 4, and 8 bytes of additional information (ai) following the initial byte in the head of the data item.

Surprisingly, Section 8.1 of RFC 8949 [STD94] does not address ai=0 to ai=23 — the assumption seems to have been 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 making this explicit:

_i ("immediate") stands for encoding with ai=0 to ai=23, i.e., it indicates that the argument is encoded directly in the initial byte of the CBOR item.

Encoding indicators are an extension point for EDN; Section 6.2 defines a registry for additional values.

Specific forms of encoding indicators are discussed in further detail in Section 2.5.4 for indefinite length strings and in Section 2.6.2 for arrays and maps.

2.4. Numbers

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

Numbers composed only of digits (of the respective base) are interpreted as CBOR integers (major type 0/1, or where the number cannot be represented in this way, major type 6 with tag 2/3). A leading "+" sign is a no-op, and a leading "-" sign inverts the sign of the number. So 0, 000, +0 all represent the same integer zero, as does -0. Similarly, 1, 001, +1 and +0001 all stand for the same integer one, and -1 and -0001 both designate the same integer minus one.

Using a decimal point (.) and/or an exponent (e for decimal, p for hexadecimal) turns the number into a floating point number (major type 7) instead, irrespective of whether it is an integral number mathematically. Note that, in floating point numbers, 0.0 is not the same number as -0.0, even if they are mathematically equal.

In Table 2, all the items on a row are the same number (also shown in CBOR, hexadecimally), but they are distinct from items in a different row.

EDN	CBOR hex
4711, 0x1267, 0o11147, 0b1001001100111	19 1267 # uint
1.5, 0.15e1, 15e-1, 0x1.8p0, 0x18p-4	F9 3E00 # float16
0, +0, -0	00 # uint
0.0, +0.0	F9 0000 # float16
-0.0	F9 8000 # float16
Infinity	F9 7C00 # float16
-Infinity	F9 FC00 # float16
NaN	F9 7E00 # float16

Table 2: Example Sets of Equivalent Notations for Some Numbers

The non-finite floating-point values Infinity, -Infinity, and NaN are written exactly as in this sentence (this is also a way they can be written in JavaScript, although JSON does not allow them). NaN in EDN stands for the NaN value with a zero sign bit and an all-zero significand except for a set quiet bit; this is represented as 0xF97E00 in CBOR Preferred Serialization. Table 3 shows how the floating point numbers 1.1, 1.5 and these three values are encoded in preferred serialization and when encoding indicators are given.

EDN	CBOR hex
1.1	fb 3ff199999999999a
1.1_1, 1.1_2	(error)
1.1_3	fb 3ff199999999999a
1.5, 1.5_1	f9 3e00
1.5_2	fa 3fc00000
1.5_3	fb 3ff8000000000000
Infinity, Infinity_1	f9 7c00
Infinity_2	fa 7f800000
Infinity_3	fb 7ff0000000000000
-Infinity, -Infinity_1	f9 fc00
-Infinity_2	fa ff800000
-Infinity_3	fb fff0000000000000
NaN, NaN_1	f9 7e00
NaN_2	fa 7fc00000
NaN_3	fb 7ff8000000000000

Table 3: Encoding indicators on floating point values

See Section 5.1, Paragraph 7, Item 3 for additional details of the EDN number syntax.

(Note that literals for further number formats, e.g., for representing rational numbers as fractions, or for other NaN values than the one called NaN, can be added as application-oriented literals. Background information beyond that in [STD94] about the representation of numbers in CBOR can be found in the informational document [I-D.bormann-cbor-numbers].)

2.5. Strings

CBOR distinguishes two kinds of strings: text strings (the bytes in the string constitute UTF-8 [STD63] text, major type 3), and byte strings (CBOR does not further characterize the bytes that constitute the string, major type 2).

EDN has three direct (unprefixed) notations for strings: double-quoted and raw strings for (UTF-8) text strings, and single-quoted strings for byte strings. The latter are useful for byte strings carrying bytes that can be meaningfully notated as UTF-8 text (Section 2.5.2).

Many strings are best notated as extension literals, which may provide detailed access to the bits within those bytes (see Section 2.5.5). Extension literals can be constructed out of single-quoted strings and raw strings, as well as sequence literals.

2.5.1. Double-Quoted String Literals

EDN enables notating text strings in a form compatible to that of notating text strings in JSON (i.e., as a double-quoted string literal), with a number of usability extensions. In JSON, no control characters are allowed to occur directly in text string literals; if needed, they can be specified using escapes such as \t or \r. In EDN, string literals additionally can contain newlines (LINEFEED U+000A), which are copied into the resulting string like other characters in the string literal. To deal with variability in platform presentation of newlines, any carriage return characters (U+000D) that may be present in the EDN string literal are not copied into the resulting string (see Section 5.1, Paragraph 7, Item 2). No other control characters can occur directly in a string literal, and the handling of escaped characters (\r etc.) is as in JSON.

JSON's escape scheme for characters that are not on Unicode's basic multilingual plane (BMP) is cumbersome (see Section 7 of RFC 8259 [STD90]). EDN keeps it, but also adds the syntax \u{NNN} where NNN is the Unicode scalar value as a hexadecimal number. This means the following are equivalent (the first o is escaped as \u{6f} for no particular reason):

```
"D\u{6f}mino's \u{1F073} + \u{2318}"   # \u{}-escape 3 chars
"Domino's \uD83C\uDC73 + \u2318"       # escape JSON-like
"Domino's  + "                       # unescaped
```

2.5.2. Single-Quoted String Literals

Analogously to text string literals delimited by double quotes, EDN allows the use of single quotes (without a prefix) to express byte string literals with UTF-8 text; for instance, the following are equivalent:

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

The escaping rules of JSON strings are applied equivalently for text-based byte string literals, e.g., \\ stands for a single backslash and \' stands for a single quote. However, to facilitate parsing, in single-quoted strings EDN excludes certain escaping mechanisms available for double-quoted strings:

* \/ is an escape in JSON that is available for EDN text strings as well to ensure all JSON texts are EDN literals. Since EDN's single-quoted strings do not occur in JSON, this legacy compatibility feature is not available for them.

* \u-based escapes are not available for characters in the range from U+0020 to U+007E (essentially, printable ASCII).

Single-quoted string literals can occur unprefixed and stand for the byte string that encodes its text string value (the "content"), or be prefixed by what looks like an application-extension prefix (see Section 2.1).

In a prefixed string literal, the text content of the single-quoted string literal is not used directly as a byte string, but is further processed in a way that is defined by the meaning given to the prefix. Depending on the prefix, the result of that processing can, but need not be, a byte string value.

Prefixed string literals (whether single-quoted after the prefix or a raw string (Section 2.5.3)) are used both for base-encoded byte string literals (see Section 2.5.5) and for application-oriented extension literals (see Section 2.1, called app-string). (Additional kinds of base-encoded string literals can be defined as application-oriented extension literals by registering their prefixes; there is no fundamental difference between the two predefined base-encoded string literal prefixes (h, b64) and any such potential future extension literal prefixes; for simplicity of expression, both cases are referred to as "extension literals".)

2.5.3. Raw String Literals

Both double-quoted and single-quoted string literals handle backslashes in a special way. For string data items that employ backslashes themselves, possibly with additional layers of processing giving this "escaping" mechanism specific application semantics, this can lead to an exponential duplication of backslashes that has informally been described as "quoting hell".

EDN therefore also allows text strings to be notated as raw string literals, which do not perform backslash processing. Instead, data transparency is provided by enclosing them in starting and ending delimiters built as a sequence of one or more backquote (‘, U+0060 GRAVE ACCENT) characters.

For example, the I-Regexp [^ \t\n\r"‘], a character class that excludes blank space and quoting characters, can be notated as:

```
 ‘‘[^ \t\n\r"‘]‘‘
```

instead of

```
 "[^ \\t\\n\\r\"‘]"
```

By using more backquotes for the outer delimiters than the longest sequence of backquotes that can be found in the string, internal backquotes do not prematurely end the string literal. An example for a raw string that contains a double backquote and therefore is notated starting and ending with a triple backquote:

```
‘‘‘To emulate typographic quotes, sometimes duplicate backward and
forward single quotes are used, as in ‘‘text.’’
’’’
```

This mechanism is easy to use for the large majority of cases. However:

* Raw strings cannot be used for empty string data items, which therefore need to be notated using double- or single-quoted strings. (Obviously, there is no need to escape the content of empty strings, so this should not be a problem.)

* Without additional rules, raw strings could not be used for string data items that start or end with backquotes, as these would amalgamate with the start and end delimiters.

To address the latter cases, two additional rules are added:

* After processing the backquotes used as delimiters, any single newline at the start of a raw string is removed. As a result:

  ‘‘‘a’’’

  can also be expressed as

  ‘‘‘
  a’’’

  In addition to enabling leading backquotes in raw strings, this can be very useful for documentation strings etc.

  This rule also allows notating ‘‘text’’ as:

  ‘‘‘
  ‘‘text’’’’’

* An ending delimiter with more backquotes than were used in the starting delimiter contributes the superfluous ones to the string.

  This allows notating a = ‘‘foo‘‘ as:

  ‘‘‘a = ‘‘foo’’’’’

(The examples given here are minimal in that they show how the additional rules work; more complex examples would be necessary to provide additional motivation why this is a good case to handle.)

See Section 5.1 for a more formal approach to defining these rules.

2.5.4. Encoding Indicators of Strings

For indefinite length encoding, strings (byte and text strings) have a special syntax streamstring. This is used (except for the special cases ’’_ and ""_ below) to notate their detailed composition into individual "chunks" (Section 3.2.3 of RFC 8949 [STD94]), by representing the individual chunks in sequence within parentheses, each optionally followed by a comma, with an encoding indicator _ immediately after the opening parenthesis: e.g., (_ h’0123’, h’4567’) or (_ "foo", "bar"). The overall type (byte string or text string) of the string is provided by the types of the individual chunks, which all need to be of the same type (Section 3.2.3 of RFC 8949 [STD94]).

For an indefinite-length string with no chunks inside, (_) would be ambiguous as to whether a byte string (encoded 0x5fff) or a text string (encoded 0x7fff) is meant and is therefore not used. The

basic forms ''_ and ""_ can be used instead and are reserved for the case of no chunks only — not as short forms for the (permitted, but not really useful) encodings with only empty chunks, which need to be notated as (_ ''), (_ ""), etc., when it is desired to preserve the chunk structure.

2.5.5. Base-Encoded Byte String Literals

Besides the unprefixed byte string literals that are analogous to JSON text string literals, EDN provides extension literals that can represent byte strings by base-encoding them, typically notated as prefixed string literals. The application-extension identifier selects one of the base encodings [RFC4648], without padding. Most often, the base encoding is enclosed in a single-quoted or raw string literal, prefixed by h for base16 or b64 for base64 or base64url (the actual encodings of the latter two have the same meaning where they overlap, so the string remains unambiguous). For example, the byte string consisting of the four bytes 12 34 56 78 (given in hexadecimal here) could be written h'12345678' or b64'EjRWeA' when using single-quoted string literals, or h‘12345678‘ or b64‘EjRWeA‘ when using raw string literals.

> (Note that Section 8 of RFC 8949 [STD94] also mentions b32 for base32 and h32 for base32hex. This has not been implemented widely and therefore is not directly included in this specification. These and further byte string formats now can easily be added back as application-oriented extension literals.)

Examples often benefit from some blank space (spaces, line breaks) in byte strings literals. In certain EDN prefixed byte string literals, blank space is ignored; for instance, the following are equivalent:

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

The internal syntax of prefixed single-quote literals such as h'' and b64'' can also allow comments as blank space (see Section 2.2).

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

Slash characters are part of the base64 classic alphabet (see Table 1 in Section 4 of [RFC4648]), and they therefore need to be in the b64'' set of characters that contribute to the byte string. Therefore, only end-of-line comments are available inside b64 byte string literals.

```
b64'/base64 not a comment/ but one follows # comment'
h'FDB6AC 7BAE27A2D69CA2699E9EDFDBBADA2779FA25 968C2C'
```

These two byte string literals stand for the same byte string; the deliberately confusing base64 content starts with b64'/bas' which is the same as h'FDB6AC' and ends with b64'lows' which is the same as h'968C2C'.

2.5.6. CBOR Sequence Literals

In diagnostic notation, a sequence of zero or more CBOR data item literals can be enclosed in << and >>, optionally prefixed by an application-extension prefix; this specification speaks of _sequence literals_. EDN mainly deals with individual data items, not with CBOR sequences [RFC8742], so the CBOR sequence represented by the sequence literal needs to be further processed to obtain the value of the literal.

Prefixed sequence literals refer to the application extension (see Section 2.1) identified by the prefix and apply the extension to its sequence content, resulting in a single data item. This data item may be a string or may not (always) be, depending on the definition of the application extension.

An unprefixed sequence literal applies CBOR encoding to the data items in its content, taken as a CBOR sequence. The value of the literal thus is a byte string with the encoded content; this is commonly referred to as _embedded CBOR_. For instance, each pair of columns in the following are equivalent:

```
<<1>>              h'01'
<<1, 2>>           h'0102'
<<"hello", null>>  h'65 68656c6c6f f6'
<<>>               h''
```

2.5.7. Validity of Text Strings

To be valid CBOR, Section 5.3.1 of RFC 8949 [STD94] requires that text strings are byte sequences in UTF-8 [STD63] form. EDN provides several ways to construct such byte strings (see Section 5.1, Paragraph 9, Item 1 for details). These mechanisms might operate on subsequences that do not themselves constitute UTF-8, e.g., by building larger sequences out of concatenating the subsequences; for validity of a text string resulting from these mechanisms it is only of importance that the result is UTF-8. Double-quoted, single-quoted, and raw string literals have been defined such that they lead to byte sequences that are UTF-8: the source language of EDN is UTF-8, and all escaping mechanisms lead only to adding further UTF-8 characters. Only prefixed string literals, other application-extensions, or in certain cases concatenation (Section 5.1, Paragraph 9, Item 1) can generate non-UTF-8 byte sequences.

As discussed at the start of Section 2, EDN implementations MAY support generation and possibly ingestion of EDN for CBOR data items that are well-formed but not valid; when this is enabled, such implementations MAY relax the requirement on text strings to be valid UTF-8.

Note that neither CBOR about its text strings nor EDN about its source language make any requirements except for conformance to [STD63]. No additional Unicode processing or validation such as normalization or checking whether a scalar value is actually assigned is foreseen by EDN, particularly not any processing that is dependent on a specific Unicode version. Such processing, if offered, MUST NOT get in the way of processing the data item represented in EDN (i.e., it may be appropriate to issue warnings but not to error out or to generate output that does not match the input at the UTF-8 level).

2.6. Arrays and Maps

EDN borrows the JSON syntax for arrays and maps. (Maps are called objects in JSON.)

2.6.1. Mandatory Separators, Optional Terminators

For maps, EDN extends the JSON syntax by allowing any data item in the map key position (before the colon).

JSON requires the use of a comma as a separator character between the elements of an array as well as between the members (key/value pairs) of a map. (These commas also were required in the original diagnostic notation defined in [STD94] and [RFC8610].) The separator commas are now optional in the places where EDN syntax allows commas;

however, where no comma is used in a separator position, there must be blank space (composed of at least one space, newline, and/or comment) instead. (Stylistically, leaving out the commas is more idiomatic when they occur at line breaks, which provide the blank space.)

In addition, 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.

In summary, the following eight examples are all equivalent:

```
[1, 2, 3]
[1, 2, 3,]
[1  2  3]
[1  2  3,]
[1  2, 3]
[1  2, 3,]
[1, 2  3]
[1, 2  3,]
```

as are

```
{1: "n", "x": "a"}
{1: "n", "x": "a",}
{1: "n"  "x": "a"}
# etc.
```

As a comma and/or blank/comment is mandatory in a separator position, [11] is unambiguously an array with a single element (the integer 11), different from [1 1] or [1,1]. As this is a general rule, [[] []] or [[],[]] are well-formed EDN, while [[][]] is not.

> | CDDL's comma separators in the equivalent 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). In summary, comma use is now aligned between EDN and CDDL, in a fully backward compatible way. (CDDL does allow the stylistically questionable a = [[][]], though.)

2.6.2. Encoding Indicators of Arrays and Maps

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

At the same position, encoding indicators for specifying the size of the array or map head for definite-length format can be used instead, specifically _i or _0 to _3. For example [_0 false, true] can be used to specify the encoding of the array [false, true] as 98 02 f4 f5.

2.6.3. Validity of Maps

As discussed at the start of Section 2, EDN implementations MAY support generation and possibly ingestion of EDN for CBOR data items that are well-formed but not valid (Section 5.3 of RFC 8949 [STD94]).

For maps, this is relevant for map keys that occur more than once, as in:

{1: "to", 1: "fro"}

2.7. Tags

A tag is written as a decimal unsigned integer for the tag number, followed by the tag content in parentheses; for instance, a date in the format specified by RFC 3339 (ISO 8601) could be notated as:

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

or the equivalent epoch-based time as the following:

```
1(1363896240)
```

The tag number can be followed by an encoding indicator giving the encoding of the tag head. For example:

```
1_1(1363896240)
```

(assuming Preferred Serialization for the tag content) is encoded as

```
d9 0001        # tag(1)
   1a 514b67b0 # unsigned(1363896240)
```

2.8. Simple values

EDN uses JSON syntax for the simple values True (true), False (false), and Null (null). Undefined is written undefined as in JavaScript.

These and all other simple values can be given as "simple()" with the appropriate unsigned integer in the parentheses. For example, simple(42) indicates major type 7, value 42, and simple(20) indicates false.

3. Application-Oriented Extension Literals

This document extends the syntax used in diagnostic notation to also enable application-oriented extensions. This section defines a number of application-oriented extensions.

3.1. 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 content of the literal is a single Standard Date/Time String as per Section 3.4.1 of RFC 8949 [STD94], as a text or byte string.

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 5), 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.

Each row of Table 4 shows an example of "dt" notation and equivalent notation not using an application-extension identifier.

dt literal	plain EDN
dt’1969-07-21T02:56:16Z’	-14159024
dt’1969-07-21T02:56:16.0Z’	-14159024.0
dt’1969-07-21T02:56:16.5Z’	-14159023.5
dt‘1969-07-21T02:56:16.5Z‘	-14159023.5
dt<<’1969-07-21T02:56:16.5Z’>>	-14159023.5
dt<<"1969-07-21T02:56:16.5Z">>	-14159023.5
dt<<‘1969-07-21T02:56:16.5Z‘>>	-14159023.5
DT’1969-07-21T02:56:16Z’	1(-14159024)

Table 4: dt and DT literals vs. plain EDN

See Section 5.2.3 for an ABNF definition for the text string input of dt literals.

3.2. 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 input of the literal is a single text string representing an IPv4address or IPv6address as per Section 3.2.2 of [RFC3986].

With the lower-case app-string prefix ip, the value of the literal is a byte string representing the binary IP address. With the upper-case app-string prefix 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 prefix IP'' can be used with an IP address 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 this application-extension provides no direct representation of the "Interface format" defined in Section 3.1.3 of [RFC9164], an address combined with an optional prefix length and an optional zone identifier, and therefore no way to reference a zone identifier at all. (If needed, this format can be put together by building their structures explicitly, e.g., an interface format without a zone identifier can be represented as in 52([ip'192.0.2.42',24]), or an interface format with zone identifier 42 as in 54([ip'fe80::0202:02ff:ffff:fe03:0303',64,42]).)

Each row of Table 5 shows an example of "ip" notation and equivalent notation not using an application-extension identifier.

ip literal	plain EDN
ip'192.0.2.42'	h'c000022a'
ip<<'192.0.2.42'>>	h'c000022a'
IP'192.0.2.42'	52(h'c000022a')
IP'192.0.2.0/24'	52([24,h'c00002'])
ip'2001:db8::42'	h'20010db8000000000000000000000042'
IP'2001:db8::42'	54(h'20010db8000000000000000000000042')
IP'2001:db8::/64'	54([64,h'20010db8'])

Table 5: ip and IP literals vs. plain EDN

See Section 5.2.4 for an ABNF definition for the content of ip literals.

3.3. The "hash" Extension

The application-extension identifier "hash" is used to notate the input to a cryptographic hash function as well as identify such a hash function to obtain a byte string that represents the output of that hash function.

The input of the literal is a (text or byte) string, optionally followed by either an integer or a text string that identifies the hash function in the COSE Algorithms registry of the CBOR Object Signing and Encryption (COSE) registry group [IANA.cose], either by the identifier (value: integer or string), or, if no algorithm is registered with this value, by its name used in the registry. If the second item is not given, the default algorithm used is -16 ("SHA-256").

No uppercase variant prefix is defined for the application-extension identifier "hash".

hash literal	plain EDN
hash<<'foo'>>	h'2C26B46B68FFC68FF99B453C1D304134 13422D706483BFA0F98A5E886266E7AE'
hash'foo'	h'2C26B46B68FFC68FF99B453C1D304134 13422D706483BFA0F98A5E886266E7AE'
hash<<'foo', -16>>	h'2C26B46B68FFC68FF99B453C1D304134 13422D706483BFA0F98A5E886266E7AE'
hash<<'foo', "SHA-256">>	h'2C26B46B68FFC68FF99B453C1D304134 13422D706483BFA0F98A5E886266E7AE'
hash<<'foo', -44>>	h'F7FBBA6E0636F890E56FBBF3283E524C 6FA3204AE298382D624741D0DC663832 6E282C41BE5E4254D8820772C5518A2C 5A8C0C7F7EDA19594A7EB539453E1ED7'
hash<<'foo', "SHA-512">>	h'F7FBBA6E0636F890E56FBBF3283E524C 6FA3204AE298382D624741D0DC663832 6E282C41BE5E4254D8820772C5518A2C 5A8C0C7F7EDA19594A7EB539453E1ED7'

Table 6: hash literals vs. plain EDN

3.4. The "cri" Extension

The application-extension identifier "cri" is used to notate an EDN literal for a CRI reference as defined in [I-D.ietf-core-href].

The input of the literal is a URI Reference as per [RFC3986] or an IRI Reference as per [RFC3987].

The value of the literal is a CRI reference that can be converted to the text of the literal using the procedure of Section 6.1 of [I-D.ietf-core-href]. Note that there may be more than one CRI reference that can be converted to the URI/IRI reference given; implementations are expected to favor the simplest variant available and make non-surprising choices otherwise. In the all-upper-case variant of the app-prefix, the value is enclosed in a tag number 99.

As an example, the CBOR diagnostic notation

```
cri'https://example.com/bottarga/shaved'
CRI'https://example.com/bottarga/shaved'
```

is equivalent to

```
[-4, ["example", "com"], ["bottarga", "shaved"]]
99([-4, ["example", "com"], ["bottarga", "shaved"]])
```

See Section 5.2.5 for an ABNF definition for the content of cri literals.

4. 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 4.1) 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 4.2).

> | Implementation note: Typically, the ultimate applications will fail if they encounter tags unknown to them, which the ones defined in this section likely are. Where chains of tools are involved in processing EDN, it may be useful to fail earlier than at the ultimate receiver in the chain unless specific processing options (e.g., command line flags) are given that indicate which of these stand-ins are expected at this stage in the chain.

4.1. 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 6.5). The content of this tag is an array of a text string for the application-extension identifier, and another array:

* For app-strings, the second array contains a single item, a text string containing the text notated by the single-quoted string in the app-string.

* For app-sequences, the second array contains zero or more items, which represent each item in the sequence contained in the app-sequence.

For example, cri’https://example.com’ can be represented as /CPA/ 999(["cri", ["https://example.com"]]), or hash<<"data", -44>> as /CPA/ 999(["hash", ["data", -44]]).

If a stage of ingestion is not prepared to handle the Unresolved Application-Extension Tag, this is an error and processing has to stop, as if this stage had been ingesting an unknown or unimplemented application-extension literal itself.

// 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.
```

4.2. 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 6.5). 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",
  "bytes_in_IRI": 'https://a.example/' + ... + '&q=bergrentrger',
  "signature": h'4711...0815',
}
```

The example "contract" combines string concatenation via the + operator (Section 5.1) with an ellipsis. 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"]),
  "bytes_in_IRI": 888(['https://a.example/', 888(null),
                       '&q=bergrentrger']),
  "signature": 888([h'4711', 888(null), h'0815']),
}
```

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

```
{ "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.

5. ABNF Definitions

This section 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 section are intended to be useful in a Parsing Expression Grammar (PEG) parser interpretation (see Appendix A of [RFC8610] for an introduction into PEG).

5.1. Overall ABNF Definition for Extended Diagnostic Notation

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

> This ABNF definition treats all single-quoted string literals the same, whether they are unprefixed and constitute byte string literals, or prefixed and their content subject to further processing. The text string value of the single-quoted strings that goes into that further processing is described using separate ABNF definitions in Section 5.2; as a convention, the grammar for the content of an app-string with prefix, say, p, is described by an ABNF definition with the rule name app-string-p.
>
> As an implementation note, some implementations may want to integrate the parsing and processing of app-string content for certain application extensions with the overall grammar. Example grammars for such integrated parsers are provided with this specification in Section 5.3.

For simplicity, the internal parsing for the built-in EDN prefixes is specified in the same way. ABNF definitions for h''/h'' and b64''/b64'' are provided in Section 5.2.1 and Section 5.2.2. However, the prefixes b32'' and h32'' are not in wide use and an ABNF definition in this document would therefore not have been based on implementation experience.

```
seq             = S [item *(MSC item) SOC]
one-item        = S item S
item            = map / array / tagged
                / number / simple
                / string / streamstring

string1         = (tstr / bstr) spec
string1e        = string1 / ellipsis
ellipsis        = 3*"." ; "..." or more dots
```

```
   string          = string1e *(S "+" S string1e)

   number          = (hexfloat / hexint / octint / binint
                      / decnumber / nonfin) spec
   sign            = "+" / "-"
   decnumber       = [sign] (1*DIGIT ["." *DIGIT] / "." 1*DIGIT)
                            ["e" [sign] 1*DIGIT]
   hexfloat        = [sign] "0x" (1*HEXDIG ["." *HEXDIG] / "." 1*HEXDIG)
                            "p" [sign] 1*DIGIT
   hexint          = [sign] "0x" 1*HEXDIG
   octint          = [sign] "0o" 1*ODIGIT
   binint          = [sign] "0b" 1*BDIGIT
   nonfin          = %s"Infinity"
                   / %s"-Infinity"
                   / %s"NaN"
   simple          = %s"false"
                   / %s"true"
                   / %s"null"
                   / %s"undefined"
                   / %s"simple(" S simple-number S ")"
   simple-number   = "25" %x30-35          ; 250-255
                   / "2" %x30-34 DIGIT     ; 200-249
                   / "1" 2DIGIT            ; 100-199
                   / %x34-39 DIGIT         ; 40-99
                   / "3" %x32-39           ; 32-39
                   ;; there are no simple values between 24-31
                   / "2" %x30-33           ; 20-23
                   / "1" DIGIT             ; 10-19
                   / DIGIT                 ; 0-9
   uint            = "0" / DIGIT1 *DIGIT
   tagged          = uint spec "(" S item S ")"

   app-prefix      = lcalpha *lcldh ; including h and b64
                   / ucalpha *ucldh ; tagged variant, if defined
   app-string      = app-prefix sqstr
   app-sequence    = app-prefix "<<" seq ">>"
   app-rstring     = app-prefix rawstring
   rawstring       = startrawdelim
                     [newline] ; swallow up to one leading newline
                     rawcontent
                     matchrawdelim
   rawdelim        = 1*"`"
   startrawdelim   = rawdelim
                     ; width (number of backquotes) distinguishes
                     ; between following matchrawdelim and shortrawdelim
   matchrawdelim   = rawdelim ; width >= previous startrawdelim
   shortrawdelim   = rawdelim ; width < previous startrawdelim
   rawchars        = 1*(%x0a/%x0d / %x20-5f / %x61-7e / NONASCII)
```

```
   rawcontent      = 1*(rawchars / shortrawdelim)

   sqstr           = SQUOTE *single-quoted SQUOTE
   bstr            = app-string / sqstr / app-rstring / rawstring
                   / app-sequence / embedded
                     ; note: rawstring is text; app-... can be any type
   tstr            = DQUOTE *double-quoted DQUOTE
   embedded        = "<<" seq ">>"

   array           = "[" (specms S item *(MSC item) SOC / spec S) "]"
   map             = "{" (specms S keyp *(MSC keyp) SOC / spec S) "}"
   keyp            = item S ":" S item

   ; We allow %x09 HT in prose, but not in string literals
   blank           = %x09 / %x0A / %x0D / %x20
   lblank          = %x0A / %x20  ; Not HT or CR (gone)
   non-slash       = blank / %x21-2e / %x30-7F / NONASCII
   non-slash-star  = blank / %x21-29 / %x2b-2e / %x30-7F / NONASCII
   non-star        = blank / %x21-29 / %x2b-7F / NONASCII
   ends-in-star    = *non-star 1*"*"
   non-lf          = %x09 / %x0D / %x20-7F / NONASCII
   eol-comment     = "#" / "//"
   comment         = "/" non-slash-star *non-slash "/"
                   / "/*" ends-in-star
                          *(non-slash-star ends-in-star) "/"
                   / eol-comment *non-lf %x0A
   ; optional space
   S               = *blank *(comment *blank)
   ; mandatory space
   MS              = (blank/comment) S
   ; mandatory comma and/or space
   MSC             = ("," S) / (MS ["," S])
   ; optional comma and/or space
   SOC             = S ["," S]

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

   double-quoted   = unescaped
                   / SQUOTE
                   / "\" escapable-d

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

```
   escapable1      = %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
                   / "\"   ; \ backslash (reverse solidus) U+005C

   escapable-d     = escapable1
                   / DQUOTE
                   / "/"   ; / slash (solidus) U+002F (JSON!)
                   / (%s"u" hexchar) ;  uXXXX      U+XXXX

   escapable-s     = escapable1
                   / SQUOTE
                   / (%s"u" hexchar-s) ;  uXXXX      U+XXXX

   hexchar         = "{" (1*"0" [ hexscalar ] / hexscalar) "}"
                   / non-surrogate
                   / two-surrogate
   non-surrogate   = ((DIGIT / "A"/"B"/"C" / "E"/"F") 3HEXDIG)
                   / ("D" ODIGIT 2HEXDIG )
   two-surrogate   = high-surrogate "\" %s"u" low-surrogate
   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

   ; single-quote hexchar-s: don't allow 0020..007e
   hexchar-s       = "{" (1*"0" [ hexscalar-s ] / hexscalar-s) "}"
                   / non-surrogate-s
                   / two-surrogate
   non-surrogate-s = "007F"                ; rubout
                   / "00" ("0"/"1"/"8"/"9"/HEXDIGA) HEXDIG
                   / "0" HEXDIG1 2HEXDIG
                   / non-surrogate-1
   non-surrogate-1 = ((DIGIT1 / "A"/"B"/"C" / "E"/"F") 3HEXDIG)
                   / ("D" ODIGIT 2HEXDIG )
   hexscalar-s     = "10" 4HEXDIG / HEXDIG1 4HEXDIG
                   / non-surrogate-1 / HEXDIG1 2HEXDIG
                   / ("1"/"8"/"9"/HEXDIGA) HEXDIG
                   / "7F"
                   / HEXDIG1

   ; Note that no other C0 characters are allowed, including %x09 HT
   unescaped       = %x0A ; new line
                   / %x0D ; carriage return -- ignored on input
                   / %x20-21
                        ; omit 0x22 "
```

```
                  / %x23-26
                       ; omit 0x27 '
                  / %x28-5B
                       ; omit 0x5C \
                  / %x5D-7F
                  / NONASCII

   newline        = [%x0D] %x0A
   DQUOTE         = %x22    ; " double quote
   SQUOTE         = "'"     ; ' single quote
   DIGIT          = %x30-39 ; 0-9
   DIGIT1         = %x31-39 ; 1-9
   ODIGIT         = %x30-37 ; 0-7
   BDIGIT         = %x30-31 ; 0-1
   HEXDIGA        = "A" / "B" / "C" / "D" / "E" / "F"
   ; Note: double-quoted strings as in "A" are case-insensitive in ABNF
   HEXDIG         = DIGIT / HEXDIGA
   HEXDIG1        = DIGIT1 / HEXDIGA
   lcalpha        = %x61-7A ; a-z
   lcldh          = lcalpha / DIGIT / "-"
   ucalpha        = %x41-5A ; A-Z
   ucldh          = ucalpha / DIGIT / "-"
   ALPHA          = lcalpha / ucalpha
   wordchar       = "_" / ALPHA / DIGIT ; [_a-z0-9A-Z]
   NONASCII       = %x80-D7FF / %xE000-10FFFF
```

Figure 1: Overall ABNF Definition of CBOR EDN

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:

1. As mentioned in the terminology (Section 1.2), the ABNF terminal values in this document define Unicode scalar values (characters) rather than their UTF-8 encoding. For example, the Unicode PLACE OF INTEREST SIGN (U+2318) would be defined in ABNF as %x2318.

2. Unicode CARRIAGE RETURN characters (U+000D, often seen escaped as "\r" in many programming languages) that exist in the input (unescaped) are ignored as if they were not in the input wherever they appear. This is most important when they are found in (text or byte) string contexts (see the "unescaped" ABNF rule). On some platforms, a carriage return is always added in front of a LINE FEED (U+000A, also often seen escaped as "\n" in many programming languages), but on other platforms, carriage returns

are not used at line breaks. The intent behind ignoring unescaped carriage returns is to ensure that input generated or processed on either of these kinds of platforms will generate the same bytes in the CBOR data items created from that input. (Platforms that use just a CARRIAGE RETURN by itself to signify an end of line are no longer relevant and the files they produce are out of scope for this document.) If a carriage return is needed in the CBOR data item, it can be added explicitly using the escaped form \r.

3. 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.

4. hexint, octint, and binint stand for an integer in the usual base 16/hexadecimal ("0x"), base 8/octal ("0o"), or base 2/binary ("0b") notation. hexfloat 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 6.4.4.3 of [C], or Section 5.13.4 of [Cplusplus]; floating-suffix/floating-point-suffix from the latter two is not used here).

5. For hexint, octint, binint, and when decnumber stands for an integer, the corresponding CBOR data item is represented using major type 0 or 1 if possible, or using tag 2 or 3 if not. In the latter case, this specification does not define any encoding indicators that apply. If fine control over encoding is desired, this can be expressed by being explicit about the representation as a tag: E.g., 98765432109876543210, which is equivalent to 2(h'35 8a 75 04 38 f3 80 f5 f6') in its Preferred Serialization, might be written as 2_3(h'00 00 00 35 8a 75 04 38 f3 80 f5 f6'_1) if leading zeros need to be added during serialization to obtain specific sizes for tag head, byte string head, and the overall byte string.

   When decnumber stands for a floating point value, and for hexfloat and nonfin, a floating point data item with major type 7 is used in Preferred Serialization (unless modified by an encoding indicator, which then needs to be _1, _2, or _3). For this, the number range needs to fit into an [IEEE754] binary64 (or the size corresponding to the encoding indicator), and the precision will be adjusted to binary64 before further applying

Preferred Serialization (or to the size corresponding to the encoding indicator). Tag 4/5 representations are not generated in these cases. Future app-prefixes could be defined to allow more control for obtaining a tag 4/5 representation directly from a hex or decimal floating point literal.

6. spec stands for an encoding indicator. See Section 2.3 for details.

7. The ABNF grammar for raw strings is lenient; a parser needs to implement the ABNF comments on matchrawdelim and shortrawdelim as well. shortrawdelim only matches sequences of backquotes that are shorter than startrawdelim. matchrawdelim only matches sequences of backquotes that are as long or longer than startrawdelim. Any excess number of backquotes in matchrawdelim are added to the string content.

   | In a PEG parser that implements predicates, these matching
   | rules can for instance be implemented as follows:
   |
   | startrawdelim = rawdelim&{|(rd)|@rdlen = rd.text_value.length}
   | shortrawdelim = rawdelim&{|(rd)|rd.text_value.length < @rdlen}
   | matchrawdelim = rawdelim&{|(rd)|rd.text_value.length >= @rdlen}

8. Extended diagnostic notation allows a (text or byte) string to be built up from multiple (text or byte) string literals, separated by a + operator; these are then concatenated into a single string.

   string, string1e, string1, and ellipsis realize: (1) the representation of strings in this form split up into multiple chunks, and (2) the use of ellipses to represent elisions (Section 4.2).

   Text strings and byte strings do not mix within such a concatenation, except that byte string literal notation can be used inside a sequence of concatenated text string notation literals, to encode characters that may be better represented in an encoded way. The following four text string values (adapted from Appendix G.4 of [RFC8610] by updating to explicit + operators) are equivalent:

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

   Similarly, the following byte string values are equivalent:

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

The semantic processing of these constructs is governed by the following rules:

* A single ... is a general ellipsis, which by itself can stand for any data item. Multiple adjacent concatenated ellipses are equivalent to a single ellipsis.

* An ellipsis can be concatenated (on one or both sides) with string chunks (string1); 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).

* If there is no ellipsis in the concatenated list, the result of processing the list will always be a single item.

* The bytes in the concatenated sequence of string chunks are simply joined together, proceeding from left to right. If the left hand side of a concatenation is a text string, the joining operation results in a text string, and that result needs to be valid UTF-8 except for implementations that support and are enabled for generation/ingestion of EDN for CBOR data items that are well-formed but not valid. If the left hand side is a byte string, the right hand side also needs to be a byte string.

* Some of the strings may be app-strings. If the result type of the app-string is an actual (text or byte) string, joining of those string chunks occurs as with chunks directly notated as string literals; otherwise the occurrence of more than one app-string or an app-string together with a directly notated string cannot be processed. (This determination must be made at the time the app-string is interpreted; see Section 4.1 for how this may not be immediately during parsing.)

5.1.1. Discussion

Note that the syntax defined here for concatenation of components uses an explicit + operator between the components to be concatenated.

> | This is not entirely backward compatible to Appendix G.4 of
> | [RFC8610], which used simple juxtaposition to indicate
> | concatenation of strings. This was not widely implemented and
> | got in the way of making the use of commas optional in other
> | places via the rule OC.

5.2. ABNF Definitions for Application Extension Content

This subsection provides ABNF definitions for the content of application-oriented extension literals defined in [STD94] and in this specification, where applicable. These grammars describe the _decoded_ content of the sqstr components that combine with the application-extension identifiers used as prefixes to form application-oriented extension literals. Each of these may integrate ABNF rules defined in Figure 1, which are not always repeated here.

Table 7 summarizes the app-prefix values defined in this document.

app-prefix	content of single-quoted string	result type
h	hexadecimal form of binary data	byte string
H	(not used)	
b64	base64 forms (classic or base64url) of binary data	byte string
B64	(not used)	
dt	RFC 3339 date/time	number (int or float)
DT	"	Tag 1 on the above
ip	IP address or prefix	byte string, array of length and byte string
IP	"	Tag 54 (IPv6) or 52 (IPv4) on the above
hash	string (usually used with sequences)	byte string
cri	RFC 3986 URI or URI reference	CBOR structure representing equivalent CRI
CRI	"	Tag 99 on the above

Table 7: App-prefix Values Defined in this Document

Note that implementation platforms may already provide implementations of grammars used in application-extensions, such as of RFC 3339 for dt'' and of IP address syntax for ip''. EDN-based tools may want to use these implementation libraries instead of using the grammars that are provided here as a reference.

For convenience, the common definitions in Figure 2 are not repeated in the below ABNF grammars.

```
ALPHA           = %x41-5a / %x61-7a
DIGIT           = %x30-39 ; 0-9
HEXDIG          = DIGIT / HEXDIGA
HEXDIGA         = "A" / "B" / "C" / "D" / "E" / "F"
; Note: double-quoted strings as in "A" are case-insensitive in ABNF
lblank          = %x0A / %x20  ; Not HT or CR (gone)
non-lf          = %x20-7f / NONASCII
NONASCII        = %x80-D7FF / %xE000-10FFFF
```

Figure 2: Common Rules Used in app-extension ABNF grammars

5.2.1. 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 3. This syntax accommodates both lower case and upper case hex digits, as well as blank space (including comments) around each hex digit.

```
app-string-h    = S *(HEXDIG S HEXDIG S / ellipsis S)
                  [eol-comment *non-lf]
ellipsis        = 3*"."
non-slash       = lblank / %x21-2e / %x30-7f / NONASCII
non-slash-star  = lblank / %x21-29 / %x2b-2e / %x30-7f / NONASCII
non-star        = lblank / %x21-29 / %x2b-7f / NONASCII
ends-in-star    = *non-star 1*"*"
non-lf          = %x20-7f / NONASCII
eol-comment     = "#" / "//"
S               = *lblank *(comment *lblank)
comment         = "/" non-slash-star *non-slash "/"
                / "/*" ends-in-star
                       *(non-slash-star ends-in-star) "/"
                / eol-comment *non-lf %x0A
```

Figure 3: ABNF Definition of Hexadecimal Representation of a Byte String

```
|  The comment syntax provided inside the hex string is intended
|  to mimic the overall syntax for comments in EDN (Section 2.2).
|  Implementation note: Comments and blank space are also
|  described by the following search regexp, which can be used to
|  remove them.  For display, the regexp is split along the outer
|  alternative into four lines, which need to be combined before
|  use; \z stands for the end of the string and is notated $ in
|  some regexp dialects.
|
|       \s|
|       /\*(?:[^*]*\*+)(?:[^/*][^*]*\*+)*/|
|       /[^/*][^/]*/|
|       (?:#|//)[^\n]*(?:\n|\z)
```

5.2.2. b64: ABNF Definition of Base64 representation of a byte string

The syntax of the content of byte strings represented in base64 is described by the ABNF in Figure 4.

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 comments based on slash characters in b64, as "/" is valid base64-classic.

```
app-string-b64  = B *(4(b64dig B))
                  [b64dig B b64dig B ["=" B "=" / b64dig B ["="]] B]
                  ["#" *non-lf]
b64dig          = ALPHA / DIGIT / "-" / "_" / "+" / "/"
B               = *lblank *(comment *lblank)
comment         = "#" *non-lf %x0A
```

Figure 4: ABNF definition of Base64 Representation of a Byte String

5.2.3. 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 in Figure 5. This is derived from [RFC3339] as summarized in Section 3 of [RFC9165].

```
   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
```

Figure 5: ABNF Definition of RFC3339 Representation of a Date/Time

5.2.4. 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 6.

```
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 ":"   ls32
              / [ 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

DIGIT1        = %x31-39 ; 1-9
uint          = "0" / DIGIT1 *DIGIT
```

Figure 6: ABNF Definition of Textual Representation of an IP Address

5.2.5. cri: ABNF Definition of URI Representation of a CRI

It can be expected that implementations of the application-extension identifier "cri" will make use of platform-provided URI implementations, which will include a URI parser.

In case such a URI parser is not available or inconvenient to integrate, a grammar of the content of cri literals is provided by the ABNF for URI-reference in Section 4.1 of RFC 3986 [RFC3986] with certain re-arrangements taken from Section 5.2.4; these are reproduced in Figure 7. If the content is not ASCII only (i.e., for IRIs), first apply Section 3.1 of [RFC3987] and apply this grammar to the result.

```
   app-string-cri = URI-reference
   ; ABNF from RFC 3986:

   URI           = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

   hier-part     = "//" authority path-abempty
                    / path-absolute
                    / path-rootless
                    / path-empty

   URI-reference = URI / relative-ref

   absolute-URI  = scheme ":" hier-part [ "?" query ]

   relative-ref  = relative-part [ "?" query ] [ "#" fragment ]

   relative-part = "//" authority path-abempty
                    / path-absolute
                    / path-noscheme
                    / path-empty

   scheme        = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )

   authority     = [ userinfo "@" ] host [ ":" port ]
   userinfo      = *( unreserved / pct-encoded / sub-delims / ":" )
   host          = IP-literal / IPv4address / reg-name
   port          = *DIGIT

   IP-literal    = "[" ( IPv6address / IPvFuture  ) "]"

   IPvFuture     = "v" 1*HEXDIG "." 1*( unreserved / sub-delims / ":" )

   ; Use IPv6address, h16, ls32, IPv4adress, dec-octet as re-arranged
   ; for PEG Compatibility in Figure 6 of [RFC XXXX]:

   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 ":"   ls32
                 / [ 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

   reg-name      = *( unreserved / pct-encoded / sub-delims )

   path          = path-abempty    ; begins with "/" or is empty
                    / path-absolute   ; begins with "/" but not "//"
                    / path-noscheme   ; begins with a non-colon segment
                    / path-rootless   ; begins with a segment
                    / path-empty      ; zero characters

   path-abempty  = *( "/" segment )
   path-absolute = "/" [ segment-nz *( "/" segment ) ]
   path-noscheme = segment-nz-nc *( "/" segment )
   path-rootless = segment-nz *( "/" segment )
   path-empty    = 0<pchar>

   segment       = *pchar
   segment-nz    = 1*pchar
   segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
                    ; non-zero-length segment without any colon ":"

   pchar         = unreserved / pct-encoded / sub-delims / ":" / "@"

   query         = *( pchar / "/" / "?" )

   fragment      = *( pchar / "/" / "?" )

   pct-encoded   = "%" HEXDIG HEXDIG

   unreserved    = ALPHA / DIGIT / "-" / "." / "_" / "~"
   reserved      = gen-delims / sub-delims
   gen-delims    = ":" / "/" / "?" / "#" / "[" / "]" / "@"
   sub-delims    = "!" / "$" / "&" / "'" / "(" / ")"
                    / "*" / "+" / "," / ";" / "="
```

Figure 7: ABNF Definition of URI Representation of a CRI

5.3. ABNF Definitions for Integrated Extension Parsers

For some applications of EDN, it is an optimization to integrate parsers for the content of some prefixed single-quoted string literals into the main parser, handling both the string literal syntax (e.g., escapes such as \' and \\) and the syntax of the extension content in one go.

For application-extensions that only use printable ASCII characters (from U+0020 to U+007E) minus single-quote ’ and backslash \, the ABNF such as that given in Section 5.2 can be directly used as an integrated parser, after adding some glue ABNF. For instance, for app-string-dt, add an alternative to bstr that points to a rule for prefixed single-quoted string literals (Figure 8).

```
bstr            = sq-app-string-dt /
                  app-string / sqstr / app-sequence / embedded
sq-app-string-dt = (%s"dt’"/%s"DT’") app-string-dt "’"
```

Figure 8: Glue ABNF for Integrated DT Parser

To facilitate writing integrated ABNF for more complex prefixed string literals, the ABNF definitions in Figure 9 may be useful and are used in the rest of this section.

```
i-HT =        %s"\t" / %s"\u" ("0009" / "{" *("0") "9}")
i-LF = %x0a / %s"\n" / %s"\u" ("000A" / "{" *("0") "A}")
i-CR = %x0d / %s"\r" / %s"\u" ("000D" / "{" *("0") "D}")

i-blank = i-LF / i-CR / " "
i-non-lf = i-HT / i-CR / %x20-26 / "\’" / %x28-5b
         / "\\" / %x5d-7f / i-NONASCII

i-NONASCII = NONASCII / %s"\u" ESCGE7F

; hex escaping for U+007F or greater
ESCGE7F = "D" ("8"/"9"/"A"/"B") 2HEXDIG
          %s"\u" "D" ("C"/"D"/"E"/"F") 2HEXDIG
        / FOURHEX1 / "0" HEXDIG1 2HEXDIG / "00" TWOHEX1
        / "{" *("0")
          ("10" 4HEXDIG / HEXDIG1 4HEXDIG
           / FOURHEX1 / HEXDIG1 2HEXDIG / TWOHEX1)
          "}"

; xxxx - 0xxx - Dhigh\uDloow
FOURHEX1 = (DIGIT1 / "A"/"B"/"C" / "E"/"F") 3HEXDIG
         / "D" ODIGIT 2HEXDIG
; 00xx - ASCII + 007F
TWOHEX1  = ("8"/"9" / HEXDIGA) HEXDIG / "7F"
```

Figure 9: ABNF Definitions Useful for Integrated Extension Parsers

Similarly, for integrated parsers for extension literals built from raw strings, the ABNF definitions in Figure 10 can be useful. fitrawdelim only matches sequences of backquotes that are exactly as long as a previous startrawdelim.

```
fitrawdelim  = rawdelim ; width == previous startrawdelim
r-non-lf = %x0D / %x20-5f / %x61-7f / NONASCII / shortrawdelim
```

Figure 10: ABNF Definitions Useful for Raw String Integrated Extension Parsers

> | In a PEG parser that implements predicates, the matching rule for fitrawdelim can for instance be implemented as follows:
> |
> | fitrawdelim = rawdelim&{|(rd)|rd.text_value.length == @rdlen}

Four subsections with ABNF for integrated parsers follow, a pair for h'' and b64'', and a pair for h'' and b64''. There is no requirement for a new application-extension to supply ABNF for an integrated parser (or any ABNF at all!), in particular if the parsing function is likely to be fulfilled by a platform library. If ABNF for the content of a single-quoted string is available in an application-extension specification, ABNF for an integrated parser can be written as a separate activity or also automatically derived. At the time of writing, one example for a tool performing such a derivation is available as open-source software [ABNFROB].

5.3.1. h'': ABNF Definition of Integrated Parser

With glue ABNF similar to that in Figure 8 and common definitions in Figures 2 and 9, ABNF such as that shown in Figure 11 can be used as an integrated parser for h prefixed single-quote strings.

```
sq-app-string-h = %s"h'" s-app-string-h "'"
s-app-string-h = h-S *(HEXDIG h-S HEXDIG h-S / ellipsis h-S)
    [eol-comment *i-non-lf]

h-S = *(i-blank) *(h-comment *(i-blank))
h-non-slash = i-blank / %x21-26 / "\'" / %x28-2e
            / %x30-5b / "\\" / %x5d-7f / i-NONASCII
h-non-slash-star = i-blank / %x21-26 / "\'" / %x28-29 / %x2b-2e
                 / %x30-5b / "\\" / %x5d-7f / i-NONASCII
h-non-star = i-blank / %x21-26 / "\'" / %x28-29 / %x2b-5b
           / "\\" / %x5d-7f / i-NONASCII
h-ends-in-star = *h-non-star 1*"*"
h-comment = "/" h-non-slash-star *h-non-slash "/"
          / "/*" h-ends-in-star
                 *(h-non-slash-star h-ends-in-star) "/"
          / eol-comment *i-non-lf i-LF
```

Figure 11: ABNF Definition for Integrated Hex Parser

5.3.2. b64'': ABNF Definition of Integrated Parser

With glue ABNF similar to that in Figure 8 and common definitions in Figures 2 and 9, ABNF such as that shown in Figure 12 can be used as an integrated parser for b64 prefixed single-quote strings.

```
sq-app-string-b64 = %s"b64'" s-app-string-b64 "'"
s-app-string-b64  = b64-S *(4(b64dig b64-S))
                  [b64dig b64-S b64dig b64-S
                   ["=" b64-S "=" / b64dig b64-S ["="]] b64-S]
                  ["#" *i-non-lf]
b64dig          = ALPHA / DIGIT / "-" / "_" / "+" / "/"
b64-S           = *i-blank *(b64-comment *i-blank)
b64-comment     = "#" *i-non-lf %x0A
```

Figure 12: ABNF Definition for Integrated Base64 Parser

5.3.3. h'': ABNF Definition of Integrated Parser

With glue ABNF similar to that in Figure 8 and common definitions in Figures 2, 9 and 10, ABNF such as that shown in Figure 13 can be used as an integrated parser for h prefixed raw strings.

```
raw-app-string-h = %s"h" startrawdelim r-app-string-h
r-app-string-h = rh-S *(HEXDIG rh-S HEXDIG rh-S / ellipsis rh-S)
    (eol-comment *r-non-lf matchrawdelim / fitrawdelim)
rh-S = *(lblank) *(rh-comment *(lblank))
rh-2 = %x61-7f / NONASCII / shortrawdelim
rh-non-slash = lblank / %x21-2e / %x30-5f / rh-2
rh-non-slash-star = lblank / %x21-29 / %x2b-2e / %x30-5f / rh-2
rh-non-star = lblank / %x21-29 / %x2b-5f / rh-2
rh-ends-in-star = *rh-non-star 1*"*"
rh-comment = "/" rh-non-slash-star *rh-non-slash "/"
           / "/*" rh-ends-in-star
                  *(rh-non-slash-star rh-ends-in-star) "/"
           / eol-comment *r-non-lf %x0A
```

Figure 13: ABNF Definition for Integrated Raw String Hex Parser

5.3.4. b64'': ABNF Definition of Integrated Parser

With glue ABNF similar to that in Figure 8, common definitions in Figures 2, 9 and 10 as well as the rule b64dig from Figure 12, ABNF such as that shown in Figure 14 can be used as an integrated parser for b64 prefixed raw strings.

```
raw-app-string-b64 = %s"b64" startrawdelim r-app-string-b64
r-app-string-b64  = rb64-S *(4(b64dig rb64-S))
                  [b64dig rb64-S b64dig rb64-S
                   ["=" rb64-S "=" / b64dig rb64-S ["="]] rb64-S]
                  ("#" *r-non-lf matchrawdelim / fitrawdelim)
rb64-S           = *lblank *(rb64-comment *lblank)
rb64-comment     = "#" *r-non-lf %x0A
```

Figure 14: ABNF Definition for Integrated Raw String Base64 Parser

6. 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.
```

6.1. 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, digits, and hyphens 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 8; all initial entries have the Change Controller "IETF".

Application-extension Identifier	Description	Reference
h	Reserved	RFC8949
b32	Reserved	RFC8949
h32	Reserved	RFC8949
b64	Reserved	RFC8949
false	Reserved	RFC-XXXX
true	Reserved	RFC-XXXX
null	Reserved	RFC-XXXX
undefined	Reserved	RFC-XXXX
dt	Date/Time	RFC-XXXX
ip	IP Address/ Prefix	RFC-XXXX
hash	Cryptographic Hash	RFC-XXXX
cri	Constrained Resource Identifier	RFC-XXXX, [I-D.ietf-core-href]

Table 8: Initial Content of Application-extension Identifier Registry

6.2. 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]).

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 9; all initial entries have the Change Controller "IETF".

Encoding Indicator	Description	Reference
_	Indefinite Length Encoding (ai=31)	RFC8949, RFC-XXXX
_i	ai=0 to ai=23	RFC-XXXX
_0	ai=24	RFC8949, RFC-XXXX
_1	ai=25	RFC8949, RFC-XXXX
_2	ai=26	RFC8949, RFC-XXXX
_3	ai=27	RFC8949, RFC-XXXX
_4	Reserved (for ai=28)	RFC-XXXX
_5	Reserved (for ai=29)	RFC-XXXX
_6	Reserved (for ai=30)	RFC-XXXX
_7	Reserved (see _)	RFC8949, RFC-XXXX

Table 9: Initial Content of Encoding Indicator Registry

| As the "Reference" column reflects, all the encoding indicators initially registered are already defined in Section 8.1 of RFC 8949 [STD94], with the exception of _i, which is defined in Section 5.1 of the present document.

6.3. Media Type

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

Name	Template	Reference
cbor-diagnostic	application/cbor-diagnostic	RFC-XXXX, Section 6.3

Table 10: 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 7 of RFC XXXX
Interoperability considerations: none
Published specification: Section 6.3 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

6.4. 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:

Content-Type	Content Coding	ID	Reference
application/cbor-diagnostic	-	TBD1	RFC-XXXX

Table 11: New Content-Format for application/cbor-diagnostic

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.

6.5. 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 12 from the "specification required" space (suggested assignments: 888 and 999), with the present document as the specification reference.

Tag	Data Item	Semantics	Reference
CPA888	null or array	Diagnostic Notation Ellipsis	RFC-XXXX
CPA999	array	Diagnostic Notation Unresolved Application-Extension	RFC-XXXX

Table 12: Values for Tags

7. Security considerations

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

The EDN specification provides two explicit extension points, application-extension identifiers (Section 6.1) and encoding indicators (Section 6.2). Extensions introduced this way can have their own security considerations (see, e.g., Section 5 of [I-D.ietf-cbor-edn-e-ref]). When implementing tools that support the use of EDN extensions, the implementer needs to be careful not to inadvertently introduce a vector for an attacker to invoke extensions not planned for by the tool operator, who might not have considered security considerations of specific extensions such as those posed by their use of dereferenceable identifiers (Section 6 of [I-D.bormann-t2trg-deref-id]). For instance, tools might require explicitly enabling the use of each extension that is not on an allowlist. This task can possibly be made less onerous by combining it with a mechanism for supplying any parameters controlling such an extension.

8. References

8.1. Normative References

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

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

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

[BCP26] Best Current Practice 26, <https://www.rfc-editor.org/info/bcp26>. At the time of writing, this BCP comprises the following:

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

[C] International Organization for Standardization, "Information technology — Programming languages — C", Edition 5, ISO/IEC 9899:2024, October 2024, <https://www.iso.org/standard/82075.html>. The standard is widely known as C23. Its technical content is also available via https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf).

[Cplusplus] International Organization for Standardization, "Programming languages — C++", Edition 7, ISO/IEC 14882:2024, October 2024, <https://www.iso.org/standard/83626.html>. The standard is widely known as C++23. Its technical content is also available via https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf (https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf).

[I-D.ietf-core-href] Bormann, C. and H. Birkholz, "Constrained Resource Identifiers", Work in Progress, Internet-Draft, draft-ietf-core-href-30, 21 November 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-core-href-30>.

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

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

[IANA.cose] IANA, "CBOR Object Signing and Encryption (COSE)", <https://www.iana.org/assignments/cose>.

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

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2019, DOI 10.1109/IEEESTD.2019.8766229, <https://ieeexplore.ieee.org/document/8766229>.

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

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

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, <https://www.rfc-editor.org/rfc/rfc3987>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/rfc/rfc7405>.

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

[RFC9164] Richardson, M. and C. Bormann, "Concise Binary Object Representation (CBOR) Tags for IPv4 and IPv6 Addresses and Prefixes", RFC 9164, DOI 10.17487/RFC9164, December 2021, <https://www.rfc-editor.org/rfc/rfc9164>.

[RFC9485] Bormann, C. and T. Bray, "I-Regexp: An Interoperable Regular Expression Format", RFC 9485, DOI 10.17487/RFC9485, October 2023, <https://www.rfc-editor.org/rfc/rfc9485>.

[STD63] Internet Standard 63, <https://www.rfc-editor.org/info/std63>. At the time of writing, this STD comprises the following:

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

[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, <https://www.rfc-editor.org/info/rfc5234>.

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

Cerf, V., "ASCII format for network interchange", STD 80,
RFC 20, DOI 10.17487/RFC0020, October 1969,
<https://www.rfc-editor.org/info/rfc20>.

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

Bormann, C. and P. Hoffman, "Concise Binary Object
Representation (CBOR)", STD 94, RFC 8949,
DOI 10.17487/RFC8949, December 2020,
<https://www.rfc-editor.org/info/rfc8949>.

8.2. Informative References

[ABNFROB] "PEG-parsing using ABNF grammars (via treetop)", n.d.,
<https://github.com/cabo/abnftt>.

[EDN-WIKI] "EDN Wiki", n.d., <https://github.com/cbor-wg/edn/wiki>.

[I-D.bormann-cbor-numbers]
Bormann, C., "On Numbers in CBOR", Work in Progress,
Internet-Draft, draft-bormann-cbor-numbers-03, 1 March
2026, <https://datatracker.ietf.org/doc/html/draft-
bormann-cbor-numbers-03>.

[I-D.bormann-t2trg-deref-id]
Bormann, C. and C. Amsss, "The "dereferenceable
identifier" pattern", Work in Progress, Internet-Draft,
draft-bormann-t2trg-deref-id-07, 24 February 2026,
<https://datatracker.ietf.org/doc/html/draft-bormann-
t2trg-deref-id-07>.

[I-D.ietf-cbor-edn-e-ref]
Bormann, C., "External References to Values in CBOR
Diagnostic Notation (EDN)", Work in Progress, Internet-
Draft, draft-ietf-cbor-edn-e-ref-03, 1 March 2026,
<https://datatracker.ietf.org/doc/html/draft-ietf-cbor-
edn-e-ref-03>.

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

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

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

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

[RFC9165] Bormann, C., "Additional Control Operators for the Concise Data Definition Language (CDDL)", RFC 9165, DOI 10.17487/RFC9165, December 2021, <https://www.rfc-editor.org/rfc/rfc9165>.

[RFC9290] Fossati, T. and C. Bormann, "Concise Problem Details for Constrained Application Protocol (CoAP) APIs", RFC 9290, DOI 10.17487/RFC9290, October 2022, <https://www.rfc-editor.org/rfc/rfc9290>.

[RFC9512] Polli, R., Wilde, E., and E. Aro, "YAML Media Type", RFC 9512, DOI 10.17487/RFC9512, February 2024, <https://www.rfc-editor.org/rfc/rfc9512>.

[RFC9682] Bormann, C., "Updates to the Concise Data Definition Language (CDDL) Grammar", RFC 9682, DOI 10.17487/RFC9682, November 2024, <https://www.rfc-editor.org/rfc/rfc9682>.

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

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

[YAML] Ben-Kiki, O., Evans, C., and I. dt Net, "YAML Ain't Markup Language (YAML) Version 1.2", Revision 1.2.2, 1 October 2021, <https://yaml.org/spec/1.2.2/>.

Appendix A. EDN and CDDL

This appendix is for information.

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 breaks are not of interest); it also adds traditional C-style inline comments (/* ... */) and end-of-line comments that start with # or //.

  EDN:

```
{ / alg / 1: -7 / ECDSA 256 / }
,
{ 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 [RFC9682]) allows specifying the actual tag number separately, while EDN's tag syntax is a simple decimal number and a pair of parentheses.

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

      CDDL:  COSE_Sign_Tagged = #6.98(COSE_Sign)
```

* 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:
         98([<< {/alg/ 1: -7 /ECDSA 256/} >>, # == h'a10126'
             ...                              # rest elided here
            ])

      CDDL:  serialized_map = bytes .cbor header_map
```

List of Figures

List of Tables

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