Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 6 August 2026

C. Bormann
Universit辰t Bremen TZI
M. G端tschow
TU Dresden

2 February 2026

Packed CBOR
draft-ietf-cbor-packed-19

Abstract

The Concise Binary Object Representation (CBOR, RFC 8949 == STD 94) 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.

CBOR does not provide any forms of data compression. CBOR data items, in particular when generated from legacy data models, often allow considerable gains in compactness when applying data compression. While traditional data compression techniques such as DEFLATE (RFC 1951) can work well for CBOR encoded data items, their disadvantage is that the recipient needs to decompress the compressed form before it can make use of the data.

This specification describes Packed CBOR, a set of CBOR tags and simple values that enable a simple transformation of an original CBOR data item into a Packed CBOR data item that is almost as easy to consume as the original CBOR data item. A separate decompression step is therefore often not required at the recipient.

// (This cref will be removed by the RFC editor:) The present
// revision -19 is a work-in-progress release in preparation for
// another cbor-packed side meeting. This revision resolves the use
// of the tunables A/B/C by setting A=16, B=8, and C=8, and choosing
// requested simple values and tag numbers, in preparation for
// continuing the early allocation process.

About This Document

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

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

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/cbor-packed.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR, [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.

CBOR does not provide any forms of data compression. CBOR data items, in particular when generated from legacy data models, often allow considerable gains in compactness when applying data compression. While traditional data compression techniques such as DEFLATE [RFC1951] can work well for CBOR encoded data items, their disadvantage is that the recipient needs to decompress the compressed form before it can make use of the data.

This specification describes Packed CBOR, a set of CBOR tags and simple values that enable a simple transformation of an original CBOR data item into a Packed CBOR data item that is almost as easy to consume as the original CBOR data item. A separate decompression step is therefore often not required at the recipient.

This document defines the Packed CBOR format by specifying the transformation from a Packed CBOR data item to the original CBOR data item; it does not define an algorithm for a packer. Different packers can differ in the amount of effort they invest in arriving at a reduced-redundancy packed form; often, they simply employ the sharing that is natural for a specific application.

Packed CBOR can make use of two kinds of optimization:

* item sharing: substructures (data items) that occur repeatedly in the original CBOR data item can be collapsed to a simple reference to a common representation of that data item. The processing required during consumption is limited to following that reference (plus carrying out integration tags (Section 5), if these are in use).

* argument sharing: application of a function with two arguments, one of which is shared. Data items (strings, containers) that share a prefix or suffix, or more generally data items that can be constructed from a function taking a shared argument and a rump data item, can be replaced by a reference to the shared argument plus a rump data item. For strings and the default "concatenation" function, the processing required during consumption is similar to following the argument reference plus that for an indefinite-length string.

A specific application protocol that employs Packed CBOR might employ both kinds of optimization or limit its use to item sharing only.

1.1. Extensibility Approach

Packed CBOR is defined in two main parts:

* Data items for referencing packing tables (Section 2), the set of which defined here which is intended to be the stable, common component of all uses of Packed CBOR, and

* Mechanisms for setting up packing tables (Section 3), which carries the main extension point, populated in this document by two table setup tags. Such setup information is usually conveyed in a tag and then applies to the content of the tag. Setup information can also be contained in environmental information that applies to an encoded CBOR data item, e.g., a media type can set up a static dictionary that applies to CBOR data items in representations that are of that media type.

Sections 4, 5, and 6 provide additional extension points, each of which is populated by one or more extensions in this document or elsewhere. These extensions can be selected by an application protocol that makes use of Packed CBOR.

Beyond the extensibility approach shown in the present document, new CBOR tags (or media types etc.) could also be defined such that they (1) modify (or completely swap out) the way the referencing data items (simple values and tags) defined in this document operate and/or (2) define new referencing data items. (From the point of view of the present specification, these tags or media types then act as setup tags setting up tables that control subtrees with semantics different from the present specification; from the point of view of the specification defining these tags or media types this simply initiates the use of the referencing data items for their specific purposes.) An example for this is not shown in the present document so that there is a coherent interpretation of the referencing data items defined here; such new definitions of referencing data items probably should specify how they interact with parts of Packed CBOR that they do not replace.

An unpacker can only carry out the tags (and the environmental information) that it knows how to interpret. An unpacker that encounters tags that are unknown to it can simply make these tags available to the application, which then can abort processing if unknown (or unimplemented) tags are found, or if their interpretation would require functionality of the unpacker that is not available. As a shortcut, the application might also provide the unpacker with a list of tags that the application can process, allowing the unpacker to abort processing when a tag unknown to it and not on this list is encountered.

1.2. Terminology and Conventions

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.

Original data item: A CBOR data item that is intended to be expressed by a packed data item; the result of all reconstructions.

Packed data item: A CBOR data item that involves packed references (_packed CBOR_).

Packed reference: A shared item reference or an argument reference,

expressed by a reference data item.

Reference data item: A data item (tag or simple value) that serves as a packed reference.

Reference site: The context of a reference data item.

Shared item reference: A reference to a shared item as defined in Section 2.2.

Argument reference: A reference that combines a shared argument with a rump item as defined in Section 2.3.

Rump: The data item contained in an argument reference that is combined with the argument to yield the reconstruction.

Straight reference: An argument reference that uses the argument as the left-hand side and the rump as the right-hand side.

Inverted reference: An argument reference that uses the rump as the left-hand side and the argument as the right-hand side.

Function tag: A tag used in an argument reference for the argument (straight references) or the rump (inverted references), causing the application of a function indicated by the function tag in order to reconstruct the data item.

Integration tag: A tag defined by an application protocol to be used as a shared item table element in order to signal a non-default procedure to integrate the shared item into the reference site.

Stand-in item: A data item (a tag or a simple value) defined by an application protocol to stand in for a more complex data item. Stand-in items are fundamentally independent of Packed CBOR but can be employed by the application protocol as part of a Packed CBOR argument reference.

Packing tables: The pair of a shared item table and an argument table.

Active set (of packing tables): The packing tables in effect at the data item under consideration.

Reconstruction: The result of applying a packed reference in the context of given packing tables; we speak of the _reconstruction of a packed reference_ as that result.

The definitions of [STD94] apply. Specifically: The term "byte" is used in its now customary sense as a synonym for "octet"; "byte strings" are CBOR data items carrying a sequence of zero or more (binary) bytes, while "text strings" are CBOR data items carrying a sequence of zero or more Unicode code points (more precisely: Unicode scalar values), encoded in UTF-8 [STD63]. In this specification, the term "argument" is not used in the specific sense assigned to it in Section 3 of RFC 8949 [STD94], but in its general sense as an argument of a function.

Where arithmetic is explained, this document uses the notation familiar from the programming language C, except that

* ".." denotes a range that includes both ends given,

* in the HTML and PDF forms, subtraction and negation are rendered as a hyphen ("-", as are various dashes), and

* superscript notation denotes exponentiation. For example, 2 to the power of 64 is notated: 2^64. In the plain-text version of this specification, superscript notation is not available and therefore is rendered by a surrogate notation. That notation is not optimized for this RFC; it is unfortunately ambiguous with C's exclusive-or and requires circumspection from the reader of the plain-text version.

Examples of CBOR data items are shown in CBOR Extended Diagnostic Notation (Section 8 of RFC 8949 [STD94] in conjunction with Appendix G of [RFC8610]
// possibly update to [I-D.ietf-cbor-edn-literals]).

2. Packed CBOR

This section describes the packing tables, their structure, and how they are referenced.

2.1. Packing Tables

At any point within a data item making use of Packed CBOR, there is an _active set_ of packing tables that applies.

There are two packing tables in an active set:

* Shared item table

* Argument table

Without any table setup, these two tables are empty arrays. Table setup can cause these arrays to be non-empty, where the elements are (potentially themselves packed) data items. Each of the tables is indexed by an unsigned integer (starting from 0). Such an index may be derived from information in tags and their content as well as from CBOR simple values.

Table setup mechanisms (see Section 3) may include all information needed for table setup within the packed CBOR data item, or they may refer to external information. This external information may be immutable, or it may be intended to potentially grow over time. In the latter case, the table setup mechanism needs to define how both backward and forward compatibility is addressed, e.g., how a reference to a new item should be handled when the unpacker uses an older version of the external information.

If, during unpacking, an index is used that references an item that is unpopulated in (e.g., outside the size of) the table in use, this MAY be treated as an error by the unpacker and abort the unpacking.

Alternatively, the unpacker MAY provide an implementation specific value, enclosed in the tag 1112, to the application and leave the error handling to the application. In the simplest case, this could be 1112(undefined), using the simple value >undefined< as per Section 5.7 of RFC 8949 [STD94]; however, the same value cannot be used repeatedly as a map key within the same map.

An unpacker SHOULD document which of these two alternatives has been chosen. CBOR based protocols that include the use of packed CBOR MAY require that unpacking errors are tolerated in some positions.

2.2. Referencing Shared Items

Shared items are stored in the shared item table of the active set.

The shared data items are referenced by using the reference data items in Table 1. The table index (an unsigned integer) is derived either from the simple value number or the (unsigned or negative) integer N provided as the content of tag 6. When reconstructing the original data item, such a reference is replaced by the referenced data item, which is then recursively unpacked.

Reference	Table Index
Simple value 0..15	0..15
Tag 6(N) (unsigned integer N 0)	16 + 2× N
Tag 6(N) (negative integer N < 0)	16 − 2× N − 1

Table 1: Referencing Shared Values

As examples, the first 22 elements of the shared item table are referenced by simple(0), simple(1), ... simple(15), 6(0), 6(-1), 6(1), 6(-2), 6(2), 6(-3). (The alternation between unsigned and negative integers for even/odd table index values — "zigzag encoding" — makes systematic use of shorter integer encodings first.)

Taking into account the encoding of these referring data items, there are 16 one-byte references, 48 two-byte references, 464 three-byte references, 130560 four-byte references, etc. As CBOR integers can grow to very large (or very negative) values, there is no practical limit to how many shared items might be used in a Packed CBOR item.

Note that the semantics of Tag 6 depend on its tag content: An integer turns the tag into a shared item reference, whereas an array of an integer and a data item turns it into an argument reference (Section 2.3). All other forms of arguments for Tag 6 are reserved for future updates to the present specification. Note also that the tag content of Tag 6 may itself be packed, so it may need to be unpacked to make this determination.

2.3. Referencing Argument Items

The argument table serves as a common table that can be used for argument references, i.e., for concatenation as well as references involving a function tag.

When referencing an argument, a distinction is made between straight and inverted references; if no function tag is involved, a straight reference combines a prefix out of the argument table with the rump data item, and an inverted reference combines a rump data item with a suffix out of the argument table.

Straight Reference	Table Index
Tag 128..135(rump)	0..7
Tag 6([unsigned integer N, rump]) (N >= 0)	8 + N

Table 2: Straight Referencing (e.g., Prefix) Arguments

Inverted Reference	Table Index
Tag 136..143(rump)	0..7
Tag 6([negative integer N, rump]) (N < 0)	8 - N - 1

Table 3: Inverted Referencing (e.g., Suffix) Arguments

Argument data items are referenced by using the reference data items in Table 2 and Table 3.

For the 16 tags 128 to 143 included, the table index (an unsigned integer) is derived from the tag number, together with the information if the reference is straight or inverted. For tag 6, the table index is derived from the integer N in the first element of the tag content (unsigned integer for straight, negative integer for inverted references). The "rump item" is the second element of the two-element array that is the tag content.

When reconstructing the original data item, such a reference is replaced by a data item constructed from the argument data item found in the table (argument, which might need to be recursively unpacked first) and the rump data item (rump, again possibly needing to be recursively unpacked).

Separate from the tag used as a reference, a tag ("function tag") may be involved to supply a function to be used in resolving the reference. It is crucial not to confuse reference tag and, if present, function tag.

A straight reference uses the argument as the provisional left-hand side and the rump data item as the provisional right-hand side. An inverted reference uses the rump data item as the provisional left-hand side and the argument as the provisional right-hand side.

In both cases, the provisional left-hand side is examined. If it is a tag ("function tag"), it is "unwrapped": The function tag's tag number is used to indicate the function to be applied, and the tag content (which, again, might need to be recursively unpacked) is kept as the unwrapped left-hand side. If the provisional left-hand side is not a tag, it is kept as the final left-hand side, and the function to be applied is concatenation, as defined below.

The following procedure applies to the data items of both the provisional right-hand side and the unwrapped left-hand side (if applicable), independent of each other: If the data item is one of the explicitly allowed stand-in items (Section 6), the item that the stand-in item stands for is recursively unpacked. If the resulting unpacked data item is again an allowed stand-in item, the previous step is repeated. If the data item is neither a stand-in item, nor further unpackable, it is taken as the final right-hand or left-hand side, respectively.

If a function tag was given, the reference is replaced by the result of applying the indicated unpacking function with the final left-hand side as its first argument and the final right-hand side as its second. The unpacking function is defined by the definition of the tag number supplied. If that definition does not define an unpacking function, the result of the unpacking is not valid.

If no function tag was given, the reference is replaced by the final left-hand side "concatenated" with the final right-hand side, where concatenation is defined as in Section 2.4.

As a contrived (but short) example, if the argument table is ["foobar", h'666f6f62', "fo"], each of the following straight (prefix) references will unpack to "foobart": 128("t"), 129("art"), 130("obart") (the byte string h'666f6f62' == 'foob' is concatenated into a text string, and the last example is not an optimization).

Taking into account the encoding, there are 8 two-byte references, 24 three-byte references, 224 four-byte references, 65280 five-byte references, etc. The numbers for inverted (suffix) references are the same. (As CBOR integers can grow to very large (or very negative) values, there is no practical limit to how many argument items might be used in a Packed CBOR item.)

2.4. Concatenation

The concatenation function is defined as follows:

* If both left-hand side and right-hand side are arrays, the result of the concatenation is an array with all elements of the left-hand-side array followed by the elements of the right-hand side array.

* If both left-hand side and right-hand side are maps, the result of the concatenation is a map that is initialized with a copy of the left-hand-side map, and then filled in with the members of the right-hand side map, replacing any existing members that have the same key. In order to be able to remove a map entry from the left-hand-side map, as a special case, any members to be replaced with a value of undefined (0xf7) from the right-hand-side map are instead removed, and right-hand-side members with the value undefined are never filled in into the concatenated map.

  | NOTES:
  |
  | * One application of the rule for straight references is to supply default values out of a dictionary, which can then be overridden by the entries in the map supplied as the rump data item.
  |
  | * Special casing the member value undefined makes it impossible to use this construct for updating maps by insertion of or replacement with actual undefined member values; undefined as a member value on the left-hand-side map stays untouched though. This exception is similar to the one JSON Merge Patch [RFC7396] makes for null values, which are however much more commonly used and therefore more problematic.

* If both left-hand side and right-hand side are one of the string types (not necessarily the same), the bytes of the left-hand side are concatenated with the bytes of the right-hand side. Byte strings concatenated with text strings need to contain valid UTF-8 data. The result of the concatenation gets the type of the unwrapped rump data item; this way a single argument table entry can be used to build both byte and text strings, depending on what type of rump is being used.

* If one side is one of the string types, and the other side is an array, the result of the concatenation is equivalent to the application of the "join" function (Section 4.1) to the string as the left-hand side and the array as the right-hand side. The original right-hand side of the concatenation determines the string type of the result.

* Other type combinations of left-hand side and right-hand side are not valid.

2.5. Discussion

This specification uses up a number of Simple Values and Tags, in particular one of the rare one-byte tags and a good chunk of the one-byte simple values. Since the objective is reduced bulk, this is warranted only based on a consensus that this specific format could be useful for a wide area of applications, while maintaining reasonable simplicity in particular at the side of the consumer. Instead of evolving the set of reference data items, this specification derives its evolvability from treating the table setup mechanism as an extension point, which can in effect provide evolved semantics to the reference data items as they reference the table.

A maliciously crafted Packed CBOR data item might contain a reference loop. A consumer/unpacker MUST protect against that.

| Different strategies for decoding/consuming Packed CBOR are available.
| For example:
|
| * the decoder can decode and unpack the packed item, presenting an unpacked data item to the application. In this case, the onus of dealing with loops is on the decoder. (This strategy generally has the highest memory consumption, but also the simplest interface to the application.) Besides avoiding getting stuck in a reference loop, the decoder will need to control its resource allocation, as data items can "blow up" during unpacking.
|
| * the decoder can be oblivious of Packed CBOR. In this case, the onus of dealing with loops is on the application, as is the entire onus of dealing with Packed CBOR.
|
| * hybrid models are possible, for instance: The decoder builds a data item tree directly from the Packed CBOR as if it were oblivious, but also provides accessors that hide (resolve) the packing. In this specific case, the onus of dealing with loops is on the accessors.
|

> | In general, loop detection can be handled similarly to how
> | loops of symbolic links are handled in a file system: A system-
> | wide limit (often set to a value permitting some 20 to 40
> | indirections for symbolic links) is applied to any reference
> | chase.
>
> | NOTE: The present specification does nothing to help with the
> | packing of CBOR sequences [RFC8742]; possibly the integration
> | tag 1115 could be used in a top-level position to express CBOR
> | sequence semantics via a packed array.

2.6. Allocation

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

These specification parameters allow the current specification to be precise while the quantitative allocation discussion is ongoing. They will be replaced by specific chosen numbers when the present specification is finalized.

The sense of the WG has been to be more conservative in allocating CBOR resources to Packed CBOR than previous drafts of this document were. In addition, Section 1.1 provides a liberal way to use the allocations for reference data items in packed items that use different unpacking mechanisms from the ones described in this document.

16 1+0 simple values are allocated to shared item references. During early development of CBOR, when the bit allocation and thus the ranges of simple values were originally defined, a range of 16 allocations was kept aside for item sharing. The allocations for 1+0 simple values were therefore performed from the top of the range down, i.e., with the block of false/true/null/undefined being originally assigned to 24..27 (after the introduction of indefinite length encoding, 20..23). No further allocation has been performed in this range in the 12 years since.

8 1+1 tags are allocated to straight argument references, and 8 further 1+1 tags allocated to inverted argument references.

A single 1+0 tag is allocated to data item references beyond the above, with the tag content deciding which of the above reference mechanisms is to be used.

Note the nature of Packed CBOR means that all these allocations can be used for pretty much unlimited purposes by simply defining another table setup mechanism (media type or table-building tag).

3. Table Setup

The reference data items described in Section 2 assume that packing tables have been set up.

By default, both tables are empty (zero-length arrays).

Table setup can happen in one of two ways:

* By the application environment, e.g., a media type. These can define tables that amount to a static dictionary that can be used in a CBOR data item for this application environment. Note that, without this information, a data item that uses such a static dictionary can be decoded at the CBOR level, but not fully unpacked. The table setup mechanisms provided by this document are defined in such a way that an unpacker can at least recognize if this is the case.

* By one or more _table-building_ tags enclosing the packed content. Each tag is usually defined to build an augmented table by adding to the packing tables that already apply to the tag, and to apply the resulting augmented table when unpacking the tag content. Usually, the semantics of the tag will be to prepend items to one or more of the tables. (The specific behavior of any such tag, in the presence of a table applying to it, needs to be carefully specified.)

  Note that it may be useful to leave a particular efficiency tier alone and only prepend to a higher tier; e.g., a tag could insert shared items at table index 16 and shift anything that was already there further along in the array while leaving index 0 to 15 alone. Explicit additions by tag can combine with application-environment supplied tables that apply to the entire CBOR data item.

  Reference data items in the newly constructed (low-numbered) parts of the table are usually interpreted in the number space of that table (which includes the, now higher-numbered, inherited parts), while reference data items in any existing, inherited (higher-numbered) part continue to use the (more limited) number space of the inherited table.

Where external information is used in a table setup mechanism that is not immutable, care needs to be taken so that, over time, references to existing table entries stay valid (i.e., the information is only extended), and that a maximum size of this information is given. This allows an unpacker to recognize references to items that are not yet defined in the version of the external reference that it uses, providing backward and possibly limited (degraded) forward compatibility.

For table setup, the present specification only defines two simple table-building tags, which operate by prepending to the (by default empty) tables.

| Additional tags can be defined for dictionary referencing (possible combining that with Basic Packed CBOR mechanisms). The desirable details are likely to vary considerably between applications. A URI-based reference would be easy to define, but might be too inefficient when used in the likely combination with an ni: URI [RFC6920].

| As a hint for implementations, an algorithm for resolving references in a scenario with nested table setup tags could be described as follows:
|
| * When chasing a reference, go upward in the data item tree.
|
| * If the next up table setup tag is not of the kind that simply prepends, switch to the alternative algorithm described by the setup tag.
|
| * If the next up table setup tag fulfills the reference (i.e., the size of the provided table is larger than the reference index), use the corresponding reference, and finish this algorithm.
|
| * Otherwise, subtract the width of the table entries added in the relevant table from the reference number and continue upwards (up into the media type, which can bequeath default tables to the CBOR items in them).

3.1. Basic Packed CBOR

Two tags are predefined by this specification for packing table setup. They are defined in CDDL [RFC8610] as in Figure 1, // assuming the allocation of tag numbers 113 ('q') and 1113 for // these tags:

```
Basic-Packed-CBOR = #6.113([[*shared-and-argument-item], rump])
Split-Basic-Packed-CBOR =
                    #6.1113([[*shared-item], [*argument-item], rump])
rump = any
shared-and-argument-item = any
argument-item = any
shared-item = any
```

Figure 1: CDDL for Packed CBOR Table Setup Tags Defined in this Document

These tags extend the two tables for shared items and for arguments that apply to the entire tag, which, unless an enclosing table setup tag or a table-setting application environment (e.g., a media type) applies, are empty tables:

Tag 113 ("Basic-Packed-CBOR"): The array given as the first element of the tag content is prepended to both the tables for shared items and for arguments.

Tag 1113 ("Split-Basic-Packed-CBOR"): The arrays given as the first and second element of the tag content are prepended individually to the tables for shared items and for arguments, respectively.

As discussed in the introduction to this section, references in the supplied new arrays use the new number space (where inherited items are shifted by the new items given), while the inherited items themselves use the inherited number space (so their semantics do not change by the mere action of inheritance).

The original CBOR data item can be reconstructed by recursively replacing shared item and argument references encountered in the rump by their reconstructions.

4. Function Tags

Function tags that occur in an argument or a rump supply the semantics for reconstructing a data item from their tag content and the non-dominating rump or argument, respectively. The present specification defines three function tags.

4.1. Join Function Tags

Tag 106 (’j’) defines the "join" unpacking function, based on the concatenation function (Section 2.4).

The join function expects an item that can be concatenated as its left-hand side, and an array of such items as its right-hand side. Joining works by sequentially applying the concatenation function to the elements of the right-hand-side array, interspersing the left-hand side as the "joiner".

An example in functional notation: join(", ", ["a", "b", "c"]) returns "a, b, c".

For a right-hand side of one or more elements, the first element determines the type of the result when text strings and byte strings are mixed in the argument. For a right-hand side of one element, the joiner is not used, and that element returned. For a right-hand side of zero elements, a neutral element is generated based on the type of the joiner (empty text/byte string for a text/byte string, empty array for an array, empty map for a map).

For an example, we assume this unpacked data item:

```
["https://packed.example/foo.html",
 "coap://packed.example/bar.cbor",
 "mailto:support@packed.example"]
```

A packed form of this using straight references could be:

```
113([[106("packed.example")],
  [128(["https://", "/foo.html"]),
   128(["coap://", "/bar.cbor"]),
   128(["mailto:support@", ""])]
])
```

Tag 105 ('i') defines the "ijoin" unpacking function, which is exactly like that of tag 106, except that the left-hand side and right-hand side are interchanged ('i').

A packed form of the first example using inverted references and the ijoin tag could be:

```
113([["packed.example"],
  [136(105(["https://", "/foo.html"])),
   136(105(["coap://", "/bar.cbor"])),
   136("mailto:support@")]
])
```

A packed form of an array with many URIs that reference SenML items from the same place could be:

```
113([[105(["coaps://[2001:db8::1]/s/", ".senml"])],
  [128("temp-freezer"),
   128("temp-fridge"),
   128("temp-ambient")]
])
```

Note that for these examples, the implicit join semantics for mixed string-array concatenation as defined in Section 2.4, Paragraph 5 actually obviate the need for an explicit join/ijoin tag; the examples do serve to demonstrate the explicit usage of the tag.

4.2. Record Function Tag

Tag 114 (’r’) defines the "record" function, which combines an array of keys with an array of values into a map.

The record function expects an array as its left-hand side, whose items are treated as key items for the resulting map, and an array of equal or shorter length as its right-hand side, whose items are treated as value items for the resulting map.

The map is constructed by grouping key and value items with equal position in the provided arrays into pairs that constitute the resulting map.

The value item array MUST NOT be longer than the key item array.

The value item array MAY be shorter than the key item array, in which case the one or more unmatched value items towards the end are treated as _absent_. Additionally, value items that are the CBOR simple value undefined (simple(23), encoding 0xf7) are also treated as absent. Key items whose matching value items are absent are not included in the resulting map.

For an example, we assume this unpacked data item:

```
[{"key0": false, "key1": "value 1", "key2": 2},
 {"key0": true, "key1": "value -1", "key2": -2},
 {"key1": "", "key2": 0}]
```

A straightforward packed form of this using the record function tag could be:

```
113([[114(["key0", "key1", "key2"])],
  [128([false, "value 1", 2]),
   128([true, "value -1", -2]),
   128([undefined, "", 0])]
])
```

A slightly more concise packed form can be achieved by manipulating the key item order (recall that the order of key/value pairs in maps carries no semantics):

```
113([[114(["key1", "key2", "key0"])],
  [128(["value 1", 2, false]),
   128(["value -1", -2, true]),
   128(["", 0])]
])
```

5. Integration Tags

Integration tags fulfill a similar purpose for shared item references as function tags do for argument references. An integration tag can be used as an element of a shared item table, supplying extended semantics on how to integrate its tag content into the context from which the shared item is referenced. A regular shared item reference can be used to reference an integration tag. (Note that the generation of an integration tag can in turn be automatic in the table setup mechanism specified by an application environment (Section 3) or a table setup tag, so the integration tag may never actually physically occur in the interchanged data.)

Application protocol specifications need to be explicit about which integration tags are in use; otherwise, the unpacker will not know whether a tag in a shared item table position is an integration tag or is intended to be shared literally. (The set of integration tags in use can also be defined as part of the table setup mechanism.)

The present specification defines one integration tag.

5.1. Splicing Integration Tag

Tag 1115, the splicing integration tag, can be used with a tag content that is an array. It specifies that the tag content is "spliced" into the surrounding array of a reference item referencing that shared item, i.e. the surrounding array is replaced by one that enumerates the elements of the shared item at the site of the shared item reference.

Example: a rump of [1, 2, 3, simple(0), 7, 8, 9], where the shared item table contains 1115([4, 5, 6]) as its first item is unpacked as [1, 2, 3, 4, 5, 6, 7, 8, 9].

Example application: Splicing integration tags could be generated implicitly in the implicit table setup defined in Section 4.1 of [I-D.lenders-dns-cbor], removing the need to allow nested arrays for names.

6. Additional Stand-in Items

Application specifications that employ Packed CBOR may also enable the use of additional "stand-in" items (tags or simple values) beyond the reference items defined by Packed CBOR. These are data items used in place of original representation items such as strings or arrays, where the tag or simple value is defined to stand for a data item that can actually be used in the position of the stand-in item. Examples would be tags such as 21 to 23 (base64url, base64, uppercase hex: Section 3.4.5.2 of RFC 8949 [STD94]) or 108 (lowercase hex: Section 2.1 of [I-D.bormann-cbor-notable-tags]), which stand for text string items but internally employ more compact byte string representations that may also be more natural as application data items.

These additional stand-in items are fundamentally independent of Packed CBOR, but they also can be used as the right-hand-side of reference items (see Section 2.3, Paragraph 11).

Note that application protocol specifications need to be explicit about which stand-in items are provided for; otherwise, inconsistent interpretations at different places in a system can lead to check/use vulnerabilities.

7. Tag Validity: Equivalence Principle

In Section 5.3.2 of RFC 8949 [STD94], the validity of tags is defined in terms of type and value of their tag content. The CBOR Tag registry ([IANA.cbor-tags] as defined in Section 9.2 of RFC 8949 [STD94]) allows recording the "data item" for a registered tag, which is usually an abbreviated description of the top-level data type allowed for the tag content.

In other words, in the registry, the validity of a tag of a given tag number is described in terms of the top-level structure of the data carried in the tag content. The description of a tag might add further constraints for the data item. But in any case, a tag definition can only specify validity based on the structure of its tag content.

In Packed CBOR, a reference data item (represented as a tag or a simple value) might be "standing in" for the actual tag content of an outer tag, or for a structural component of that. In this case, the formal structure of the outer tag’s content before unpacking usually no longer fulfills the validity conditions of the outer tag.

The underlying problem is not unique to Packed CBOR. For instance, [RFC8746] describes tags 64..87 that "stand in" for CBOR arrays (the native form of which has major type 4). For the other tags defined [RFC8746], which require some array structure of the tag content, a footnote was added:

> | [...] The second element of the outer array in the data item is a native CBOR array (major type 4) or Typed Array (one of tag 64..87)

The top-down approach to handle the "rendezvous" between the outer tag and the tag content representation (e.g., using an inner tag) does not support extensibility: any further Typed Array tags being defined do not inherit the exception granted to tag number 64..87; they would need to formally update all existing tag definitions that can accept typed arrays or be of limited use with these existing tags.

Instead, the tag validity mechanism needs to be extended by a bottom-up component: A tag (or simple value) definition needs to be able to declare that the tag can "stand in" for, (is, in terms of tag validity, equivalent to) some structure.

E.g., tag 64..87 could have declared their equivalence to the CBOR major type 4 arrays they stand in for.

> | Note that not all domain extensions to tags can be addressed using the equivalence principle: E.g., on a data model level, numbers with arbitrary exponents ([ARB-EXP], tags 264 and 265) are strictly a superset of CBOR's predefined fractional types, tags 4 and 5. They could not simply declare that they are equivalent to tags 4 and 5 as a tag requiring a fractional value may have no way to handle the extended range of tag 264 and 265.

7.1. Tag Content Equivalence

A tag or simple value definition MAY declare Tag Content Equivalence to some existing structure, under some conditions defined by that definition. This, in effect, extends all existing tag definitions that accept the named structure to accept the newly defined item under the conditions given for the Tag Content Equivalence.

A number of limitations apply to Tag Content Equivalence, which therefore should be applied deliberately and sparingly:

* Tag Content Equivalence is a new concept, which may not be implemented by an existing generic decoder. A generic decoder not implementing tag equivalence might raise tag validity errors where Tag Content Equivalence says there should be none.

* A CBOR protocol MAY specify the use of Tag Content Equivalence, effectively limiting the protocol's full use to those generic encoders that implement it. Existing CBOR protocols that do not address Tag Content Equivalence implicitly have a new variant that allows Tag Content Equivalence (e.g., to support Packed CBOR with an existing protocol). A CBOR protocol that does address Tag Content Equivalence MAY be explicit about what kinds of Tag Content Equivalence it supports (e.g., only the reference tags employed by Packed CBOR and certain table setup tags).

* There is currently no way to express Tag Content Equivalence in CDDL. For Packed CBOR, CDDL would typically be used to describe the unpacked CBOR represented by it; further restricting the Packed CBOR is likely to lead to interoperability problems. (Note that, by definition, there is no need to describe Tag Equivalence on the receptacle [outer tag] side; only for the item that declares Tag Content Equivalence.)

* The registry "CBOR Tags" [IANA.cbor-tags] currently does not have a way to record any equivalence claimed for a tag. A convention would be to alert to Tag Content Equivalence in the "Semantics (short form)" field of the registry.
// Needs to be done for the tag registrations here.

7.2. Tag Content Equivalence of Tags and Simple Values Defined in Packed CBOR

The reference data items (tags and simple values) in this specification declare their equivalence to the unpacked shared items or function results they represent.

The table setup tags 113 and 1113 declare their equivalence to the unpacked CBOR data item represented by them.

8. IANA Considerations

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

For all assignments described in this section, the "reference" column is the present document, i.e., RFCXXXX.

8.1. CBOR Tags Registry

In the registry "CBOR Tags" [IANA.cbor-tags], IANA is requested to allocate the tags defined in Table 4.

Tag	Data Item	Semantics
6	int (for shared); [int, any] (for argument)	Reference Data Item (for Packed CBOR: shared/argument)
105	concatenation item (text string, byte string, array, or map)	Packed CBOR: ijoin function
106	array of concatenation item (text string, byte string, array, or map)	Packed CBOR: join function
113	array (shared-and-argument-items, rump)	Packed CBOR: table setup
114	array	Packed CBOR: record function
128..135	any	Reference Data Item (for Packed CBOR: straight argument)
136..143	function tag or concatenation item (text string, byte string, array, or map)	Reference Data Item (for Packed CBOR: inverted argument)
1112	any	Packed CBOR: reference error
1113	array (shared-items, argument-items, rump)	Packed CBOR: table setup
1115	any	Packed CBOR: splicing integration tag

Table 4: Values for Tag Numbers

8.2. CBOR Simple Values Registry

In the registry "CBOR Simple Values" [IANA.cbor-simple-values], IANA is requested to allocate the simple values defined in Table 5.

Value	Semantics
0..15	Reference Data Item (for Packed CBOR: shared)

Table 5: Simple Values

9. Security Considerations

The security considerations of [STD94] apply.

Loops in the Packed CBOR can be used as a denial of service attack unless mitigated, see Section 2.5.

As the unpacking is deterministic, packed forms can be used as signing inputs when deterministically encoded [I-D.ietf-cbor-cde]. (Note that where external dictionaries are added to cbor-packed as in [I-D.amsuess-cbor-packed-by-reference], this requires additional consideration.)

When tables are obtained from the application environment, e.g., a media type, any evolution of the application environment (such as an update to the media type specification) needs to reliably ensure that existing references continue to unpack in the same way. Therefore, application environments that provide packing tables need to explicitly specify if these packing tables may evolve, and, if yes, provide a design for this kind of evolvability. For instance, [I-D.amsuess-cbor-packed-by-reference] provides a way to reserve entries in a packing table that can be filled in by revisions of the application environment; to avoid false unpacking, this needs to be the only update that can be applied to such a table-setting application environment.

10. References

10.1. 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>.

[I-D.ietf-cbor-edn-literals] Bormann, C., "CBOR Extended Diagnostic Notation (EDN)", Work in Progress, Internet-Draft, draft-ietf-cbor-edn-literals-19, 16 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-cbor-edn-literals-19>.

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

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

[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>.

[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>.

10.2. Informative References

[ARB-EXP] Occil, P., "Arbitrary-Exponent Numbers", Specification for Registration of CBOR Tags 264 and 265, <http://peteroupc.github.io/CBOR/bigfrac.html>.

[I-D.amsuess-cbor-packed-by-reference]
Amsss, C., "Packed CBOR: Table set up by reference", Work in Progress, Internet-Draft, draft-amsuess-cbor-packed-by-reference-04, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-amsuess-cbor-packed-by-reference-04>.

[I-D.bormann-cbor-notable-tags]
Bormann, C., "Notable CBOR Tags", Work in Progress, Internet-Draft, draft-bormann-cbor-notable-tags-14, 19 January 2026, <https://datatracker.ietf.org/doc/html/draft-bormann-cbor-notable-tags-14>.

[I-D.ietf-cbor-cde]
Bormann, C., "CBOR Common Deterministic Encoding (CDE)", Work in Progress, Internet-Draft, draft-ietf-cbor-cde-13, 13 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-cbor-cde-13>.

[I-D.lenders-dns-cbor]
Lenders, M. S., Bormann, C., Schmidt, T. C., and M. Whlisch, "A Concise Binary Object Representation (CBOR) of DNS Messages", Work in Progress, Internet-Draft, draft-lenders-dns-cbor-15, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-lenders-dns-cbor-15>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/rfc/rfc1951>.

[RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <https://www.rfc-editor.org/rfc/rfc6920>.

[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>.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, DOI 10.17487/RFC7396, October 2014, <https://www.rfc-editor.org/rfc/rfc7396>.

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

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

[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>.

Appendix A. Examples

The (JSON-compatible) CBOR data structure depicted in Figure 2, 400 bytes of binary CBOR, could be packed into the CBOR data item depicted in Figure 3, 308 bytes, only employing item sharing. With support for argument sharing and the record function tag 114, the data item can be packed into 298 bytes as depicted in Figure 4. Note that this particular example does not lend itself to prefix compression, so it uses the simple common-table setup form (tag 113).

```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Figure 2: Example original CBOR data item, 400 bytes

```
113([["price", "category", "author", "title", "fiction", 8.95,
                                                        "isbn"],
    /  0         1         2         3         4         5    6   /
     {"store": {
       "book": [
         {simple(1): "reference", simple(2): "Nigel Rees",
          simple(3): "Sayings of the Century", simple(0): simple(5)},
         {simple(1): simple(4), simple(2): "Evelyn Waugh",
          simple(3): "Sword of Honour", simple(0): 12.99},
         {simple(1): simple(4), simple(2): "Herman Melville",
          simple(3): "Moby Dick", simple(6): "0-553-21311-3",
          simple(0): simple(5)},
         {simple(1): simple(4), simple(2): "J. R. R. Tolkien",
          simple(3): "The Lord of the Rings",
          simple(6): "0-395-19395-8", simple(0): 22.99}],
       "bicycle": {"color": "red", simple(0): 19.95}}}])
```

Figure 3: Example packed CBOR data item with item sharing only, 308 bytes

```
113([[114(["category", "author",
           "title", simple(1), "isbn"]),
    /  0                     /
      "price", "fiction", 8.95],
    /  1        2         3    /
     {"store": {
       "book": [
           128(["reference", "Nigel Rees",
              "Sayings of the Century", simple(3)]),
           128([simple(2), "Evelyn Waugh",
              "Sword of Honour", 12.99]),
           128([simple(2), "Herman Melville",
              "Moby Dick", simple(3), "0-553-21311-3"]),
           128([simple(2), "J. R. R. Tolkien",
               "The Lord of the Rings", 22.99, "0-395-19395-8"])],
       "bicycle": {"color": "red", simple(1): 19.95}}}])
```

Figure 4: Example packed CBOR data item using item sharing and the record function tag, 302 bytes

The (JSON-compatible) CBOR data structure below has been packed with shared item and (partial) prefix compression only and employs the split-table setup form (tag 1113).

```
{
  "name": "MyLED",
  "interactions": [
    {
      "links": [
        {
          "href":
           "http://192.168.1.103:8445/wot/thing/MyLED/rgbValueRed",
          "mediaType": "application/json"
        }
      ],
      "outputData": {
        "valueType": {
          "type": "number"
        }
      },
      "name": "rgbValueRed",
      "writable": true,
      "@type": [
        "Property"
      ]
    },
    {
      "links": [
        {
          "href":
           "http://192.168.1.103:8445/wot/thing/MyLED/rgbValueGreen",
          "mediaType": "application/json"
        }
      ],
      "outputData": {
        "valueType": {
          "type": "number"
        }
      },
      "name": "rgbValueGreen",
      "writable": true,
      "@type": [
        "Property"
      ]
    },
    {
      "links": [
        {
          "href":
           "http://192.168.1.103:8445/wot/thing/MyLED/rgbValueBlue",
          "mediaType": "application/json"
        }
```

```
      ],
      "outputData": {
        "valueType": {
          "type": "number"
        }
      },
      "name": "rgbValueBlue",
      "writable": true,
      "@type": [
        "Property"
      ]
    },
    {
      "links": [
        {
          "href":
           "http://192.168.1.103:8445/wot/thing/MyLED/rgbValueWhite",
          "mediaType": "application/json"
        }
      ],
      "outputData": {
        "valueType": {
          "type": "number"
        }
      },
      "name": "rgbValueWhite",
      "writable": true,
      "@type": [
        "Property"
      ]
    },
    {
      "links": [
        {
          "href":
           "http://192.168.1.103:8445/wot/thing/MyLED/ledOnOff",
          "mediaType": "application/json"
        }
      ],
      "outputData": {
        "valueType": {
          "type": "boolean"
        }
      },
      "name": "ledOnOff",
      "writable": true,
      "@type": [
        "Property"
```

```
        ]
      },
      {
        "links": [
          {
            "href":
  "http://192.168.1.103:8445/wot/thing/MyLED/colorTemperatureChanged",
            "mediaType": "application/json"
          }
        ],
        "outputData": {
          "valueType": {
            "type": "number"
          }
        },
        "name": "colorTemperatureChanged",
        "@type": [
          "Event"
        ]
      }
    ],
    "@type": "Lamp",
    "id": "0",
    "base": "http://192.168.1.103:8445/wot/thing",
    "@context":
     "http://192.168.1.102:8444/wot/w3c-wot-td-context.jsonld"
  }
```

Figure 5: Example original CBOR data item, 1210 bytes

```
   1113([/shared/["name", "@type", "links", "href", "mediaType",
              /  0       1       2        3         4 /
       "application/json", "outputData", {"valueType": {"type":
            /  5               6               7 /
       "number"}}, ["Property"], "writable", "valueType", "type"],
                 /   8            9           10           11 /
      /argument/ ["http://192.168.1.10", 128("3:8445/wot/thing"),
                / 128                    129 /
      129("/MyLED/"), 130("rgbValue"), "rgbValue",
        / 130             131           132     /
      {simple(6): simple(7), simple(9): true, simple(1): simple(8)}],
        / 133 /
      /rump/ {simple(0): "MyLED",
              "interactions": [
      133({simple(2): [{simple(3): 131("Red"), simple(4): simple(5)}],
       simple(0): 132("Red")}),
      133({simple(2): [{simple(3): 131("Green"), simple(4): simple(5)}],
       simple(0): 132("Green")}),
      133({simple(2): [{simple(3): 131("Blue"), simple(4): simple(5)}],
       simple(0): 132("Blue")}),
      133({simple(2): [{simple(3): 131("White"), simple(4): simple(5)}],
       simple(0): "rgbValueWhite"}),
      {simple(2): [{simple(3): 130("ledOnOff"), simple(4): simple(5)}],
       simple(6): {simple(10): {simple(11): "boolean"}}, simple(0):
       "ledOnOff", simple(9): true, simple(1): simple(8)},
      {simple(2): [{simple(3): 130("colorTemperatureChanged"),
       simple(4): simple(5)}], simple(6): simple(7), simple(0):
       "colorTemperatureChanged", simple(1): ["Event"]}],
        simple(1): "Lamp", "id": "0", "base": 129(""),
        "@context": 128("2:8444/wot/w3c-wot-td-context.jsonld")}])
```

Figure 6: Example packed CBOR data item, 507 bytes

List of Figures

List of Tables

Acknowledgements

CBOR packing was part of the original proposal that turned into CBOR, but did not make it into [RFC7049], the predecessor of RFC 8949 [STD94]. Various attempts to come up with a specification over the years did not proceed. In 2017, Sebastian Kbisch proposed investigating compact representations of W3C Thing Descriptions, which prompted the author to come up with what turned into the present design.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Concrete Contracts.

Authors' Addresses

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

Mikolai Gtschow
TUD Dresden University of Technology
Helmholtzstr. 10
D-01069 Dresden
Germany
Email: mikolai.guetschow@tu-dresden.de