not yet
Internet-Draft
Intended status: Standards Track
Expires: November 30, 2013

C. Bormann
Universitaet Bremen TZI
P. Hoffman
VPN Consortium

May 29, 2013

Concise Binary Object Representation (CBOR)
draft-bormann-cbor-01

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on November 30, 2013.

Table of Contents

1. Introduction

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

The format defined here follows some specific design goals that are not well met by current formats. The serialization is for an extended version of the JSON data model [RFC4627]. It is important to note that this is not a proposal that the grammar in RFC 4627 be extended in general, since doing so would cause a significant backwards incompatibility with already-deployed JSON documents. Instead, this document simply defines its own data model which starts from JSON.

Appendix E lists some existing binary formats and discusses how well they do or do not fit the design objectives of CBOR.

1.1. Objectives

The objectives of the Concise Binary Object Representation (CBOR), roughly in decreasing order of importance, are:

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

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

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

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

   * An encoder and a parser need to be implementable in a very small amount of code, thus being applicable to class 1 constrained nodes as defined in [I-D.ietf-lwig-terminology].

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

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

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

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

   * "Reasonable" here is bounded by JSON as an upper bound in size, and by implementation complexity maintaining a lower bound. Using either general compression schemes or extensive bit-fiddling violates the complexity goals.

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

   * This means it must be reasonably frugal in CPU usage for both encoding and parsing. This is relevant both for constrained

nodes and for potential usage in applications with a very high volume of data.

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

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

7. The format must be extensible, with the extended data being able to be parsed by earlier parsers.

   * The format is designed for decades of use.

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

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

1.2. Terminology

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

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

This specification makes use of the following terminology:

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

Parser: A process that decodes a CBOR data item and makes it available to an application. This is also sometimes called a decoder.

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

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

Stream parser: A process that decodes a data stream and makes each of the data items in the sequence available to an application.

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

2. Specification of the CBOR Encoding

A CBOR encoded data item is structured and encoded as described in this section. For the impatient reader, the encoding is summarized in Table 4 in Appendix B.

The initial byte of each data item contains both information about the major type (the high-order 3 bits) and additional information (the low-order 5 bits). When the value of the additional information is less than 24, it is directly used as a small unsigned integer. When it is 24 to 27, the additional bytes for a variable-length integer immediately follow; the values 24 to 27 of the additional information specify that its length is a 1-, 2-, 4- or 8-byte unsigned integer, respectively. Additional information value 31 is used for indefinite length arrays and maps, described below. Additional information values 28 to 30 are reserved for future expansion.

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

A CBOR parser implementation can be based on the jump table with all 256 defined values for the initial byte (Table 4). A parser in a constrained implementation can instead use the structure of the initial byte and following bytes for more compact code (see Appendix C for a rough impression of how this could look like).

2.1. Major Types

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

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

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

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

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

Major type 4: an array of data items. Arrays are also called sequences or tuples. The array's length follows the rules for byte strings (major type 2), except that the length denotes the number of data items, not the length in bytes that the array takes

up. Items in an array do not need to all be of the same type. If the additional information is 31, it means that the array has an indefinite length; such an array is terminated by a "break" stop code, 0b111_11111. For example, an array that contains 10 items of any type would have an initial byte of 0b100_01010 (major type of 4, additional information of 10 for the length) followed by the 10 remaining items.

Major type 5: a map of pairs of data items. Maps are often also called tables, dictionaries, hashes, or objects (in JSON). A map is comprised of pairs of data items, the even-numbered ones serving as keys and the following odd-numbered ones serving as values for the key that comes immediately before it. The map's length follows the rules for byte strings (major type 2), except that the length denotes the number of pairs, not the length in bytes that the map takes up. If the additional information is 31, it means that the map has an indefinite length; such a map is terminated by a "break" stop code, 0b111_11111. For example, a map that contains 9 pairs would have an initial byte of 0b101_01001 (major type of 5, additional information of 9 for the number of pairs) followed by the 18 remaining items. The first item is the first key, the second item is the first value, the third item is the second key, and so on.

Major type 6: optional semantic tagging of other major types. See Section 2.3.

Major type 7: floating point numbers and simple data types that need no content, as well as the "break" stop code. See Section 2.2.

These eight major types lead to a simple table showing which of the 256 possible values for the initial byte of a data item are used for (Table 4).

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

2.2. Floating Point Numbers and Values with No Content

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

5-bit value	semantics
0..23	Simple value (value 0..23)
24	Simple value (value 24..255 in following byte)
25	IEEE 754 Half-Precision Float (16 bits follow)
26	IEEE 754 Single-Precision Float (32 bits follow)
27	IEEE 754 Double-Precision Float (64 bits follow)
28-30	(unallocated)
31	"break" stop code for indefinite arrays and maps

Table 1: Values for Additional Information in Major Type 7

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

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

value	semantics
0..19	(unallocated)
20	False
21	True
22	Null
23	Undefined value
24..255	(unallocated)

Table 2: Simple Values

2.3. Optional Tagging of Items

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

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

CBOR tags are truly optional, and are probably of little value in applications where the implementation creating a particular CBOR data stream and the implementation parsing that stream know the semantic meaning of each item in the stream. Their primary purpose in this specification is to define common data types such as dates. A secondary purpose it to allow optional tagging when the parser is a generic CBOR parser that might be able to benefit from hints about the content of items. Understanding the semantic tags is optional for a parser; it can just jump over the initial bytes of the tag and interpret the tagged data item itself.

Applications may use specific tags defined in the following list and/or defined by standard action or in the registry.

tag	data item	semantics
0	UTF-8 string	Standard date/time string; see Section 2.3.1
1	multiple	Epoch-based date/time; see Section 2.3.1
2	byte string	Positive bignum; see Section 2.3.2
3	byte string	Negative bignum; see Section

		2.3.2
4	array	Decimal fraction; see Section 2.3.3
5	array	Chunked byte string or UTF-8 string; see Section 2.3.4
6..20	(unallocated)	(unallocated)
21	multiple	Expected conversion to base64url encoding; see Section 2.3.5.2
22	multiple	Expected conversion to base64 encoding; see Section 2.3.5.2
23	multiple	Expected conversion to base16 encoding; see Section 2.3.5.2
24	byte string	Encoded CBOR data item; see Section 2.3.5.1
25..31	(unallocated)	(unallocated)
32	UTF-8 string	URI; see Section 2.3.5.3
33	UTF-8 string	Base64url; see Section 2.3.5.3
34	UTF-8 string	Base64; see Section 2.3.5.3
35	UTF-8 string	Regular expression; see Section 2.3.5.3
36	UTF-8 string	MIME message; see Section 2.3.5.3
37+	(unallocated)	(unallocated)

Table 3: Values for tags

2.3.1. Date and Time

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

Tag value 1 is for numerical representation of seconds relative to 1970-01-01T00:00Z in UTC time. The tagged item can be a positive or negative integer (major types 0 and 1), or a floating point number

(major type 7 with additional information 25, 26 or 27). Note that the number can be negative (time before 1970-01-01T00:00Z) and, if a floating point number, indicate fractional seconds.

2.3.2. Bignums

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

For example, the number 18446744073709551616 (2**64) is represented as 0b110_00010 (major type 6, tag 2), followed by 0b010_01001 (major type 2, length 9), followed by 0x0100000000000000000 (one byte 0x01 and eight bytes 0x00).

2.3.3. Decimal Fractions

[RFC6020] defines a decimal fraction format called decimal64, which can be used for an exact representation of decimal fractions by combining a 64-bit integer with a small negative decimal (base-10) exponent. CBOR supports a slight generalization, by allowing the use of other integer lengths than 64 bit. In CBOR this is represented as an array that contains exactly two integers: the (negative, base-10) exponent and the mantissa. For example, the number 273.15 could be represented as 0b110_00100 (major type of 6 for the tag, additional information of 4 for the type of tag), followed by 0b100_00010 (major type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00001 (major type of 1 for the first integer, additional information of 1 for the value of -2), followed by 0b000_11001 (major type of 0 for the second integer, additional information of 25 for a two-byte value), followed by 0b0110101010110011 (27315 in two bytes).

2.3.4. Chunking

If an array is enclosed in a tag with value 5, it indicates that the value of the data item is built by concatenating all items within the array. This is defined for arrays where every element is a byte string, or every element is a UTF-8 string. The chunking tag can be applied to both fixed-length and indefinite-length arrays.

2.3.5. Content Hints

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

2.3.5.1. Encoded CBOR data item

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

2.3.5.2. Expected Later Encoding for CBOR to JSON Converters

Tags 21 to 23 indicate that a byte string might require a specific encoding when interoperating with a text-based representation. These tags are useful when an encoder knows that the byte string data it is writing is likely to be later converted to a particular JSON-based usage. That usage specifies that some strings are encoded as Base64, Base64url, and so on. The encoder uses byte strings instead of doing the encoding itself to reduce the message size, to reduce the code size of the encoder, or both. The encoder does not know whether or not the converter will be generic, and therefore wants to say what it believes is the proper way to convert binary strings to JSON.

The data item following this tag can be a byte string, an array, or a map. In the latter two cases, the tag applies to all of the byte strings in the data object.

These three tag types suggest conversions to three of the base data encodings defined in [RFC4648]. Where the encoding allows the use of padding ("="), this is not used. Later tags might be defined for other data encodings of RFC 4648, or of other ways to encode binary data in strings.

2.3.5.3. Encoded Text

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

- Tag 32 is for URIs, as defined in [RFC3986];
- Tags 33 and 34 are for base64url and base64 encoded text strings, as defined in [RFC4648];
- Tag 35 is for regular expressions in PCRE/JavaScript syntax [ECMA262].

o Tag 36 is for MIME messages, as defined in [RFC2045];

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

2.4. Streaming Arrays and Maps Using Indefinite Lengths

Arrays and maps can be encoded with an indefinite length (additional information value 31) if the number of items is not known when the encoding of the array or map starts; this is often referred to as "streaming". (For streaming byte strings or UTF-8 strings, the string can also be split into chunks and embedded in an indefinite length array enclosed with the chunking tag 5.) The array or map is closed by encoding a "break" stop code. The stop code is encoded with major type 7 and additional information value 31, but is not itself a data item: it is just a syntactical feature to close the indefinite length item.

For example, assume an encoder wants to represent the abstract array [1, [2, 3], [4, 5]]. The non-streaming encoding would be 0x8301820203820405. The streaming encoding could have many values, including 0x9f018202039f0405ffff, 0x9f01820203820405ff, 0x83018202039f0405ff, and 0x83019f0203ff820405.

There is no restriction against nesting streaming arrays and maps. A "break" stop code only terminates a single array/map, so nested streaming arrays/maps need exactly as many stop codes as there are type bytes starting a streaming array/map.

3. Creating CBOR-Based Protocols

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

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

This specification puts no restrictions on CBOR-based protocols. An encoder can be capable of encoding as many or as few types of values as is required by the protocol in which it is used; a parser can be

capable of understanding as many or as few types of values as is required by the protocols in which it is used. This lack or restrictions allows CBOR to be used in extremely constrained environments.

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

3.1. CBOR in Streaming Applications

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

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

3.2. Parsing Errors

3.2.1. Enforcing Restrictions on the Value Following a Tag

Tags (Section 2.3) specify what type of data item is supposed to follow the tag; for example, the tags for positive or negative bignums are supposed to be followed by byte strings. A parser that finds a data item of the wrong type after a tag might issue a warning, might stop processing altogether, might handle the error and make the incorrectly-typed value available to the application as such, or take some other type of action.

3.2.2. Handling Unknown Simple Values and Tags

A parser that comes across a simple value Section 2.2 that it does not recognize, such as a value that was added to the IANA registry after the parser was deployed or a value that the parser chose not to implement, might issue a warning, might stop processing altogether, might handle the error by making the unknown value available to the application as such, or take some other type of action.

A parser that comes across a tag Section 2.3 that it does not recognize, such as a tag that was added to the IANA registry after

the parser was deployed or a tag that the parser chose not to implement, might issue a warning, might stop processing altogether, might handle the error and present the unknown tag value together with the contained data item to the application, might ignore the tag and simply present the contained data item only to the application, or take some other type of action.

3.2.3. UTF-8 Strings

A parser might or might not want to verify that the sequence of bytes in an UTF-8 string (major type 3) is actually valid UTF-8. If a parser attempts to validate the UTF-8 and fails, it might issue a warning, might stop processing altogether, might handle the error and present the invalid string to the application as such, or take some other type of action.

3.2.4. Incomplete CBOR data items

The representation of a CBOR data item has a specific length, determined by its initial bytes and by the structure of any data items enclosed in the data items. If less data is available in the input byte string, a parser may completely fail the decoding, or substitute the missing data and data items using an decoder-specific convention. A decoder may also implement incremental parsing, that is, parse the data item as far as it is available and present the data found so far, (such as in an event-based interface) with the option of continuing the decoding once further data are available.

For instance, if a parser is expecting a certain number of array or map entries, but it instead encounters the end of the data, it should probably issue an error and/or stop processing altogether, but it might take some other action. The same is true if it is processing what it expects to be the last pair in a map and it comes to the end of the data.

Similarly, if a parser has just seen a tag and then encounters the end of the data, it should probably issue an error and/or stop processing altogether, but it might take some other action.

3.2.5. Unknown Additional Information Values

At the time this document is written, some additional information values are undefined and reserved for future versions of this document (see Section 5.2). A parser that sees an additional information value that it does not understand should probably issue an error and/or stop processing altogether, but it might take some other action.

3.3. Numbers

For the purposes of this specification, all number representations are equivalent. This means that an encoder can encode a floating point value of 0.0 as the integer 0. It, however, also means that an application that expects to find integer values only might find floating point values if the encoder decides these are desirable, such as when the floating point value is more compact than a 64-bit integer.

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

A CBOR-based protocol designed for compactness may want to exclude specific integer encodings that are longer than necessary for the application, such as to save the need to implement 64-bit integers. There is an expectation that encoders will use the most compact integer representation that can represent a given value. However, a compact application should accept values that use a longer-than needed encoding (such as encoding "0" as 0b000_11101 followed by two bytes of 0x00) as long as the application can parse an integer of the given size.

3.4. Specifying Keys for Maps

The encoding and parsing applications need to agree on what types of keys are going to be used in maps. In applications that need to interwork with JSON-based applications, keys probably should be limited to UTF-8 strings only; otherwise, there has to be a specified mapping from the other CBOR types to Unicode characters, and this often leads to implementation errors.

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

Applications for constrained devices that have maps with fewer than 24 known keys should consider using integers because the keys can then be encoded in a single byte.

3.5. Undefined Values

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

3.6. Canonical CBOR

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

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

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

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

CBOR tags make canonicalization more difficult. The absence or presence of tags in a canonical format is determined by the optionality of the tags in the protocol. In a CBOR-based protocol that allows optional tagging anywhere, the canonical format must not allow them. In a protocol that requires tags in certain places, the tag needs to appear in the canonical format.

3.7. Generic Encoders and Parsers

A generic CBOR decoder can parse all well-formed CBOR data and present them to an application. CBOR data are well-formed if the structure of the initial bytes and the byte strings/data items implied by their values is followed and no extraneous data follows (Appendix C).

Even though CBOR attempts to minimize these cases, not all well-formed CBOR data are valid: for example, the format excludes simple values below 24 that are encoded with an extension byte. Also, specific tags may make semantic constraints that may be violated, such as by including a tag in a tag or by enclosing a byte string within a date tag. Finally, the data may be invalid, such as invalid UTF-8 strings or date strings that do not conform to [RFC3339].

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

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

4. Converting Data Between CBOR and JSON

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

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

4.1. Converting From CBOR to JSON

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

- An Integer (major type 0 or 1) becomes a JSON number.
- A byte string (major type 2) that is not embedded in a tag that specifies a proposed encoding is encoded in Base64url without padding and becomes a JSON string.
- A UTF-8 string (major type 3) becomes a JSON string. Note that JSON requires escaping certain characters (RFC 4627, section 2.5): quotation mark (U+0022), reverse solidus (U+005C), and the "C0 control characters" (U+0000 through U+001F). All other characters are copied unchanged into the JSON UTF-8 string.
- An array (major type 4) becomes a JSON array.
- A map (major type 5) becomes a JSON object. This is possible directly only if all keys are UTF-8 strings. A converter might also convert other keys into UTF-8 strings (such as by converting integers into strings containing their decimal representation); however, doing so introduces a danger of key collision.
- False (major type 7, additional information 20) becomes a JSON false.
- True (major type 7, additional information 21) becomes a JSON true.
- Null (major type 7, additional information 22) becomes a JSON null.
- A floating point value (major type 7, additional information 25 through 27) becomes a JSON number if it is finite (i.e., can be represented in a JSON number); if the value is non-finite (i.e., (positive) Infinity, -Infinity, or NaN), it is represented by the substitute value.
- Any other simple value (Major type 7, any additional information value not yet discussed) is represented by the substitute value.
- A bignum (major type 6, tag value 2 or 3) is represented by encoding its byte string in Base64url without padding and becomes

a JSON string. For tag value 3 (negative bignum), a "~" (ASCII tilde) is inserted before the base-encoded value.

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

- For all other tags (major type 6, any other tag value), the embedded CBOR item is represented as a JSON value; the tag value is ignored.

4.2. Converting From JSON to CBOR

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

- JSON numbers without fractional parts (integer numbers) are represented as integers (major types 0 and 1, possibly major type 6 tag value 2 and 3), choosing the shortest form; integers longer than an implementation-defined threshold (which is usually either 32 or 64 bits) may instead be represented as floating point values. (If the JSON was generated from a JavaScript implementation, its precision is already limited to 53 bits maximum.)

- Numbers with fractional parts are represented as floating point values. Preferably, the shortest exact floating point representation is used; for instance, 1.5 is represented in a 16-bit floating point value (not all implementations will be efficiently capable of finding the minimum form, though). There may be an implementation-defined limit to the precision that will affect the precision of the represented values. Decimal representation should only be used if that is specified in a protocol.

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

5. Future Evolution of CBOR

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

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

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

5.1. Extension Points

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

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

CBOR has three major extension points:

- the "simple" space (values in major type 7). Of the 24 efficient (and 232 slightly less efficient) values, only a small number have been allocated. Implementations receiving an unknown simple data item may be able to process it as such, given that the structure of the value is indeed simple. An IANA registry is appropriate here.

- the "tag" space (values in major type 6). Again, only a small part of the code point space has been allocated, and the space is abundant (although the early numbers are more efficient than the later ones). Implementations receiving an unknown tag can choose to simply ignore it, or to process it as an unknown tag wrapping the enclosed data item. An IANA registry is appropriate here.

- the "additional information" space. An implementation receiving an unknown additional information has no way to continue parsing, so allocating codepoints to this space is a major step. There are also very few codepoints left.

5.2. Curating the Additional Information Space

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

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

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

6. Diagnostic Notation

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

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

The diagnostic notation is based on JSON as it is defined in RFC 4627. The notation borrows the JSON syntax for numbers (integer and floating point), True, False, Null, UTF-8 strings, arrays and maps (maps are called objects in JSON; the diagnostic notation extends JSON here by allowing any data item in the key position). Undefined

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

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

or the equivalent relative time as

```
1(1363896240)
```

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

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

6.1. Encoding indicators

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

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

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

An underscore followed by a decimal digit n indicates that the preceding item (or, for arrays and maps, the item starting with the preceding bracket or brace) was encoded with an additional information value of 24+n. For example, 1.5_1 is a half precision floating point number, while 1.5_3 is encoded as double precision.

(This encoding indicator is not shown in Appendix A.) (Note that the encoding indicator "_" is thus an abbreviation of the full form "_7", which is not used.)

7. IANA Considerations

IANA will create two registries for new CBOR values. The registries will follow the rules in [RFC5226]. IANA will also allocate a new MIME media type.

7.1. Simple Values Registry

A registry called "CBOR Simple Values" will be created. The initial values are shown in Table 2.

New entries in the range 0 to 19 will be allocated by Standards Action, starting with the number 16. New entries in the range 24 to 255 will be allocated by Specification Required.

7.2. Tags Registry

A registry called "CBOR Tags" will be created. The initial values are shown in Table 3.

New entries in the range 0 to 23 will be allocated by Standards Action. New entries in the range 24 to 255 will be allocated by Specification Required. New entries in the range 256 to 18446744073709551615 will be allocated by First Come First Served. The template for First Come First Served will include point of contact and an optional field for URL to a description of the semantics of the tag; the latter can be something like an Internet-Draft or a web page.

7.3. Media Type ("MIME Type")

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

Type name: application

Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: none; CBOR is a binary format

Security considerations: Same as for the base document

Interoperability considerations: n/a

Published specification: This document

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

Additional information:
Magic number(s): n/a
File extension(s): .cbor
Macintosh file type code(s): n/a

Person & email address to contact for further information:
Carsten Bormann
cabo@tzi.org

Intended usage: COMMON

Restrictions on usage: none

Author:
Carsten Bormann
cabo@tzi.org

Change controller:
Carsten Bormann
cabo@tzi.org

TBD: Maybe add application/mmmmm+cbor for specific protocols?

8. Security Considerations

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

9. Acknowledgements

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

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

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

This document also incorporates suggestions made by many people, notably James Manger, Joe Hildebrand, Phillip Hallam-Baker, Tim Bray, and Tony Finch.

10. References

10.1. Normative References

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

10.2. Informative References

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

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

[ECMA262] European Computer Manufacturers Association, "ECMAScript Language Specification 5.1 Edition", ECMA Standard ECMA-262, June 2011, <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>.

[I-D.ietf-lwig-terminology] Bormann, C., Ersue, M., and A. Keraenen, "Terminology for Constrained Node Networks", draft-ietf-lwig-terminology-04 (work in progress), April 2013.

[MessagePack] FURUHASHI Sadayuki, "MessagePack", 2013, <http://msgpack.org/>.

[RFC0713] Haverty, J., "MSDTP-Message Services Data Transmission Protocol", RFC 713, April 1976.

[RFC2045] Freed, N. and N.S. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, December 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC6020] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.

[UBJSON] The Buzz Media, "Universal Binary JSON Specification", 2013, <http://ubjson.org/>.

Appendix A. Examples

The following table provides some CBOR encoded values in hexadecimal (right column), together with diagnostic notation for these values (left column). Note that the string "\u00fc" is one form of diagnostic notation for a UTF-8 string containing the single Unicode character U+00FC, LATIN SMALL LETTER U WITH DIAERESIS (u umlaut).

Similarly, "\u6c34" is a UTF-8 string in diagnostic notation with a single character U+6C34 (CJK UNIFIED IDEOGRAPH-6C34, often representing "water"), and "\ud800\udd51" is a UTF-8 string in diagnostic notation with a single character U+10151 (GREEK ACROPHONIC ATTIC FIFTY STATERS). (Note that all these single-character strings could also be represented in native UTF-8 in diagnostic notation, just not in an ASCII-only specification like the present one.)

Diagnostic	Encoded
0	0x00
1	0x01
10	0x0a
23	0x17
24	0x1818
25	0x1819
100	0x1864
1000	0x1903e8
1000000	0x1a000f4240
1000000000000	0x1b000000e8d4a51000
18446744073709551615	0x1bffffffffffffff
18446744073709551616	0xc249010000000000000000
-18446744073709551616	0x3bffffffffffffff
-18446744073709551617	0xc349010000000000000000
-1	0x20
-10	0x29
-100	0x3863
-1000	0x3903e7

0.0	0xf90000
-0.0	0xf98000
1.0	0xf93c00
1.1	0xfb3ff199999999999a
1.5	0xf93e00
65504.0	0xf97bff
100000.0	0xfa47c35000
3.4028234663852886e+38	0xfa7f7fffff
1.0e+300	0xfb7e37e43c8800759c
5.960464477539063e-08	0xf90001
6.103515625e-05	0xf90400
-4.0	0xf9c400
-4.1	0xfbc010666666666666
Infinity	0xf97c00
NaN	0xf97e00
-Infinity	0xf9fc00
Infinity	0xfa7f800000
NaN	0xfa7fc00000
-Infinity	0xfaff800000
Infinity	0xfb7ff0000000000000
NaN	0xfb7ff8000000000000
-Infinity	0xfbfff0000000000000
false	0xf4

true	0xf5
nil	0xf6
undefined	0xf7
simple(16)	0xf0
simple(24)	0xf818
simple(255)	0xf8ff
0("2013-03-21T20:04:00Z")	0xc074323031332d30332d32315432303a30343a30305a
1(1363896240)	0xc11a514b67b0
1(1363896240.5)	0xc1fb41d452d9ec200000
23(h'01020304')	0xd74401020304
24(h'6449455446')	0xd818456449455446
32("http://www.example.com")	0xd82076687474703a2f2f7777772e6578616d706c652e636f6d
h''	0x40
h'01020304'	0x4401020304
""	0x60
"a"	0x6161
"IETF"	0x6449455446
"\"\\"	0x62225c
"\u00fc"	0x62c3bc
"\u6c34"	0x63e6b0b4
"\ud800\udd51"	0x64f0908591
[]	0x80
[1, 2, 3]	0x83010203

[1, [2, 3], [4, 5]]	0x8301820203820405
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x98190102030405060708090a0b0c0d0e0f101112131415161718181819
{}	0xa0
{1: 2, 3: 4}	0xa201020304
{"a": 1, "b": [2, 3]}	0xa26161016162820203
["a", {"b": "c"}]	0x826161a161626163
{"a": "A", "b": "B", "c": "C", "d": "D", "e": "E"}	0xa56161614161626142616361436164614461656145
5([_ "indefin", "ite", " examp", "les:"])	0xc59f67696e646566696e6369746566206578616d70646c65733aff
[_]	0x9fff
[_ 1, [2, 3], [_ 4, 5]]	0x9f018202039f0405ffff
[_ 1, [2, 3], [4, 5]]	0x9f01820203820405ff
[1, [2, 3], [_ 4, 5]]	0x83018202039f0405ff
[1, [_ 2, 3], [4, 5]]	0x83019f0203ff820405
[_ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x9f0102030405060708090a0b0c0d0e0f101112131415161718181819ff
{_ "a": 1, "b": [_ 2, 3]}	0xbf61610161629f0203ffff

["a", {_ "b": "c"}]	0x826161bf61626163ff

TBD: add more examples?

Appendix B. Jump Table

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

TBD: check again that we have all the single-byte tags represented in the table

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

0x59	byte string (two-byte uint16_t for n, and then n bytes follow)
0x5a	byte string (four-byte uint32_t for n, and then n bytes follow)
0x5b	byte string (eight-byte uint64_t for n, and then n bytes follow)
0x60..0x77	UTF-8 string (0x00..0x17 bytes follow)
0x78	UTF-8 string (one-byte uint8_t for n, and then n bytes follow)
0x79	UTF-8 string (two-byte uint16_t for n, and then n bytes follow)
0x7a	UTF-8 string (four-byte uint32_t for n, and then n bytes follow)
0x7b	UTF-8 string (eight-byte uint64_t for n, and then n bytes follow)
0x80..0x97	array (0x00..0x17 data items follow)
0x98	array (one-byte uint8_t for n, and then n data items follow)
0x99	array (two-byte uint16_t for n, and then n data items follow)
0x9a	array (four-byte uint32_t for n, and then n data items follow)
0x9b	array (eight-byte uint64_t for n, and then n data items follow)
0x9f	array, data items follow, terminated by "break" stop code
0xa0..0xb7	map (0x00..0x17 pairs of data items follow)
0xb8	map (one-byte uint8_t for n, and then n pairs of data items follow)
0xb9	map (two-byte uint16_t for n, and then n pairs of data items follow)

0xba	map (four-byte uint32_t for n, and then n pairs of data items follow)
0xbb	map (eight-byte uint64_t for n, and then n pairs of data items follow)
0xbf	map, pairs of data items follow, terminated by "break" stop code
0xc0	Text-based date/time (data item follows, see Section 2.3.1)
0xc1	Epoch-based date/time (data item follows, see Section 2.3.1)
0xc2	Positive bignum (data item "byte string" follows)
0xc3	Negative bignum (data item "byte string" follows)
0xc4	Decimal Fraction (data item "array" follows, see Section 2.3.3)
0xc5	Chunked byte/UTF-8 string (data item "array" follows, see Section 2.3.4)
0xd5..0xd7	Expected Conversion (data item follows, see Section 2.3.5.2)
0xd8	(more tagged items, one byte and then a data item follow)
0xf4	False
0xf5	True
0xf6	Null
0xf7	Undefined
0xf9	Half-Precision Float (two-byte IEEE 754)
0xfa	Single-Precision Float (four-byte IEEE 754)
0xfb	Double-Precision Float (eight-byte IEEE 754)

0xff	"break" stop code

Table 4: Jump Table for Initial Byte

Appendix C. Pseudocode

The well-formedness of a CBOR item can be checked by the pseudo-code in Figure 1. The data is well-formed, iff:

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

The pseudo-code has the following prerequisites:

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

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

```
    }
    return true;
  }

  well_formed_indefinite(mt, breakable) {
    switch (mt) {
      case 4: while (well_formed(true)); break;
      case 5: while (well_formed(true)) well_formed(); break;
      case 7:
        if (breakable)
          return false;                // signal break out
        else fail();                   // no enclosing indefinite
      default: fail();                 // wrong mt
    }
    return true;
  }
```

Figure 1: Pseudo-Code for well-formedness check

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

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

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

Figure 2: Pseudo-code for encoding a signed integer

Appendix D. Half-precision

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

```
#include <math.h>

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

Figure 3: C code for a half-precision decoder

```
import struct
from math import ldexp

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

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

Figure 4: Python code for a half-precision decoder

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

The proposal for CBOR follows a history of binary formats that is as long as the history of computers themselves. Different formats have had different objectives. In most cases, the objectives of the format were never stated, although they can sometimes be implied by the context where the format was first used. Some formats were meant

to be universally-usable, although history has proven that no binary format meets the needs of all protocols and applications.

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

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

1. unambiguously encode common data formats from Internet standards

2. code compactness for encoder or parser

3. no schema description needed

4. reaonably compact serialization

5. applicable to constrained and unconstrained applications

6. good JSON conversion

7. extensibility

E.1. ASN.1 DER and BER

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

E.2. MessagePack

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

MessagePack has been essentially stable since it was first published around 2011; it has not yet had a transition. The evolution of MessagePack is impeded by an imperative to maintain complete backwards compatibility with existing stored data, while only few bytecodes are still available for extension. Repeated requests over

the years from the MessagePack user community to separate out binary and text strings in the encoding recently have led to an extension proposal that would leave MessagePack's "raw" data ambiguous between its usages for binary and text data. The extension mechanism for MessagePack remains unclear.

E.3. BSON

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

E.4. UBJSON

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

E.5. MSDTP: RFC 713

A very early example of a compact message format is described in [RFC0713], defined in 1976. It is included here for its historical value, not because it was ever widely used.

E.6. Conciseness On The Wire

While CBOR's design objective of code compactness for encoders and decoders is higher than its objective of conciseness on the wire, many people focus on the wire size. Table 5 shows some encoding examples for the simple nested array [1, [2, 3]]; where streaming is supported by the encoding, [_ 1, [2, 3]] (indefinite length on the outer array) is also shown.

(Entries marked with an asterisk have not been checked against an implementation and might be applying some liberty in translating the CBOR data model to that format. Corrections are appreciated.)

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

Table 5: Examples for different levels of conciseness

Authors' Addresses

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

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

Paul Hoffman
VPN Consortium

Email: paul.hoffman@vpnc.org