Concise Binary Object Representation Maint&Ext
Internet-Draft
Intended status: Informational
Expires: 2 September 2026

C. Bormann
Universit辰t Bremen TZI

1 March 2026

On Numbers in CBOR
draft-bormann-cbor-numbers-03

Abstract

The Concise Binary Object Representation (CBOR), as defined in STD 94 (RFC 8949), is a data representation format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.

Among the kinds of data that a data representation format needs to be able to carry, numbers have a prominent role, but also have inherent complexity that needs attention from protocol designers, from implementers of CBOR libraries, and from implementers of the applications that use them.

This document gives an overview over number formats available in CBOR and some notable CBOR tags registered, and it attempts to provide information about opportunities and potential pitfalls of these number formats.

// This is still rather drafty, pieced together from various components, so it has a higher level of redundancy than ultimately desired.

About This Document

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

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

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

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR), as defined in RFC 8949 [STD94], is a data representation format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.

Among the kinds of data that a data representation format needs to be able to carry, numbers have a prominent role, but also have inherent complexity that needs attention from protocol designers, from implementers of CBOR libraries, and from implementers of the applications that use them.

This document gives an overview over number formats available in CBOR and some notable CBOR tags registered, and it attempts to provide information about opportunities and potential pitfalls of these number formats.

It discusses CBOR representation of numbers in four main Sections:

* Integer Numbers (Section 2),

* IEEE 754 Floating Point Values (Section 3),

* Other Floating Point Numbers (Section 4),

* Tagged Arrays of Numbers (Section 5).

These sections will generally address considerations such as:

* Encoding efficiency (number of bytes needed), possibly processing efficiency (CPU used in processing)

* Preferred Serialization, Common Deterministic Encoding (CDE, [I-D.ietf-cbor-cde], see also [I-D.bormann-cbor-det] for more background discussion)

* Use by applications

* Interoperability considerations, potential "dark corners"

CBOR defines an interchange format for various kinds of number-like data items, some right in [STD94], some via the definition of additional tags (see also Section 4). Implementations on a specific platform will want to map the transferred data items into the data types available on the platform or new data types defined by the CBOR implementation. The ranges of the various formats provided for numbers in the interchange format are not always congruent with the platform types of a specific platform, so some design effort may be required to define a platform-specific API. Apart from a relatively strong dividing line between integer and floating point data items, the interchanged data items make no statement on which platform-specific type should be used to ingest the item into. Designers of CBOR-based protocols can decide to define their application data types with a view to platform types that are likely to be available on the target platforms.

1.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [BCP14] when, and only when, they appear in all capitals, as shown here.

Terms and definitions from [STD94], [RFC8610], and [IEEE754] apply.

The somewhat stilted term "floating point datum" from [IEEE754] (a superset of "floating point number") is usually expressed as "floating point value" . (Note that RFC8949 is not always very precise in this, using the term "floating point number" almost as a synonym for "floating point value".)

2. Integer Numbers

CBOR provides representations of integer numbers in unsigned and negative forms:

* Unsigned integers up to 2^64−1, major type 0

* Negative integers down to −2^64, major type 1

* Unsigned integers with no size limitations, tag 2 on a byte string

* Negative integers with no size limitations, tag 3 on a byte string

The latter two forms are often called "bignums" for historical reasons, contrasting them to the former "basic" integers; we'll try to avoid the term as it can be confused with platform-specific types such as BigInt in JavaScript. The Concise Data Definition Language (CDDL) [RFC8610] has the types uint, nint, and int, for the ranges of values covered by major type 0, major type 1, and either of them, respectively; biguint, bignint, and bigint for the range of value covered by tag 2, tag3, and either; and unsigned and integer for a choice of either form (but interestingly no negative). As the preferred encoding for an integer chooses between major type 0/1 and tag 2/3 automatically, in practice biguint and unsigned are the same type, as are bigint and integer.

Applications that want to constrain the ranges of numbers in a way less dependent on CBOR serialization specifics can find useful CDDL number and number range definitions in [USEFUL-CDDL].

The Major type 0 numbers come in five different encoding sizes, as indicated by their initial byte: immediate ("1+0") encoding (0..23), one-byte ("1+1") (0..255), two-byte ("1+2", 0..65535), four-byte, and eight-byte. The Preferred Serialization always uses the shortest of the major type 0 encodings available for an unsigned integer. The intention is that there is no semantic difference between the different major type 0 encodings of the same value, and there also is no semantic difference between major type 0 and tag 2. This means that Preferred Serialization always uses major type 0 over tag 2 when possible, and the shortest encoding of these (and thus no leading zero bytes for the tagged encodings). Major type 1 and tag 3 are analogous.

Note that there is no "signed type" in CBOR: as any specific number to be represented is either negative or not, it is represented as an unsigned integer or as a negative integer. Major type 0 unsigned integers cover exactly the range of widely used platform types such as uint64_t or u64. Signed platform types such as int64_t or i64 can be represented in the lower half of the unsigned space and the upper half of the negative space. Platforms typically have no nint64_t type that could take all negative numbers representable in major type 1; generic decoders will therefore treat the lower half of the negative space in the same way they will treat tag 2/3 values that do

not fit the signed platform type. Similarly, generic encoders for a platform with u128/i128 types will choose between major type 0/1 and tag 2/3 just like they would choose between the encoding sizes inside major type 0/1.

While additional representation of integers could be developed, the options already provided by [STD94] should be able to satisfy most applications.

3. IEEE 754 Floating Point Values

While integer numbers are relatively easy to represent, floating point numbers as a realization of rational or real numbers are a much more varied subject. Many rational or real numbers require rounding until they can be encoded as a floating point number.

There are many choices that can be made when designing a machine representation for floating point numbers. After decades of vendor-specific formats, IEEE standardized [IEEE754], initially in 1985, updated in 2008 and then 2019 (IEC 559 is then mirroring IEEE 754). This standard is widely adopted in hardware and software, offering choices such as binary vs. decimal floating point numbers, and different representation sizes. Out of the large choice available, CBOR directly uses the three formats binary16, binary32, and binary64, i.e., the signed binary floating point formats in 16, 32, and 64 bits, colloquially known as half (16 bits), single (32 bits), and double (64 bits) precision. Most platforms that support floating point computation support at least single precision, except for the most constrained ones also double precision, while half precision is mostly used for storage and interchange only and may be software-supported only.

3.1. Integer vs. Floating Point

Mathematically speaking, integer numbers are a subset of the rational or real numbers from which floating point numbers are drawn. In many programming environments, however, integer numbers are clearly separated from floating point numbers (the most notable exception being the original JavaScript language, which only had one number type).

For specific applications, it may be desirable to represent all numbers that can be represented as integers as such, even if they are used where floating point numbers are used for non-integers. [I-D.mcnally-deterministic-cbor] defines application-level deterministic representation rules that can be used with CDE to enforce this for a certain subset of the integers.

Most CBOR applications so far have tended to get by with the kind of strong separation between the integer and floating point worlds that programming environments usually favor, so our focus will not be on approaches for intermingling them in this document.

3.2. Considerations for non-finite numbers and non-numbers

IEEE754 distinguishes three sets of floating point data item (termed _floating point datum_ in [IEEE754], also simply called _floating point value_ in CBOR-related documents), not all of which are termed _floating point numbers_:

* finite floating-point number: A finite number that is representable in a floating-point format. Note that these further divide into zero, subnormal, and normal; this distinction is usually not of interest in interchange, except that there are a few platforms with limited floating point support that may not support subnormal numbers.

* infinite floating-point number: One of the two values −Infinite and (positive) Infinite. On many platforms, infinite numbers can be accessed via a floating point operation such as 1.0/0.0 (positive infinity) or −1.0/0.0 (negative infinity); they react to comparisons as one would expect.

* NaN: a _floating point datum_ that is not a number (NaN), used to represent computations that did not lead to a numeric result, not even an infinity. A commonly implemented example for such a computation is 0.0/0.0. The formats provide a way to include additional information with a NaN, such as its sign bit, whether operations on the NaN are intended to fail immediately (signaling) or just return another NaN (quiet), and some remaining bits that may carry additional information (intended as diagnostic).

  It can be surprising that according to [IEEE754], NaN values always compare as different even if they have the same NaN information (i.e., are identical). (There is also a totalorder relation that does give NaNs a defined place, depending on their sign bits; this only recently has been standardized as part of std::strong_order in C++20 [Cplusplus20].)

Not all platforms that can use IEEE 754 do provide all these sets (or all elements of all sets), e.g., Erlang only provides finite floating-point numbers. Platforms that do provide them widely vary in the way they provide access to non-finite numbers and NaNs beyond the floating point operations given above. Usually there is an operation such as isnan() in C, which is needed as comparison of any floating point value (including NaNs) to a NaN always yields inequality.

3.2.1. Protocol Design Considerations

CBOR does not attempt to invent a new floating point architecture, but adopts the architecture defined in [IEEE754] in a whole-sale fashion.

CBOR supports the interchange of all kinds of IEEE 754 data items, including non-finite numbers and non-numbers (NaNs).

For an application developer that is already using IEEE 754 floating point, there is little additional consideration required: Both infinities and NaN are widely supported in IEEE-754 hardware and software by CPUs, OS's and programming environments. CBOR protocol designs can generally rely on infinities and NaN as a concept being supported, but implementations may run into dark corners of their platforms when it comes to distinguishing and preserving NaN information in NaN values.

However, for a protocol that wants to achieve good interoperability over a wide variety of platforms, the fact that platforms differ in their support of non-finite numbers and NaNs becomes relevant. (See Section 3.2.2 below for reasons for such differences.) Protocol designs aiming for the widest possible platform support may want to implement replacements for infinite numbers and NaNs, or at least not rely on NaN information being successfully preserved during interchange.

3.2.1.1. Use Cases for Full Support of NaN Values

Given the dark corners mentioned, a CBOR implementer might question whether there even are good use cases for full NaN support. The author conjectures that the main use case will be moving existing complex algorithms from a monolithic implementation to a distributed one. A data representation format is generally needed for this, and CBOR may be a good choice if the target environment is not entirely homogeneous. Since existing algorithms may make use of IEEE 754 including its NaN values, full support for inserting CBOR intermediate representation into an algorithm implies full support for NaNs. (This also means that the main objective for such

implementations is full data transparency, not necessary great internal APIs for dealing with the data — IEEE 754 is that API!)

3.2.1.2. JSON Compatibility

Note that JSON supports neither infinite numbers nor NaN. For protocols that are intended to work in both CBOR and JSON representations and need an out-of-band indicator comparable to NaN, a protocol developer might consider this (in CDDL, where float is not intended to be a NaN value):

```
float-with-null = float / null
```

Additional choices can be added for the infinities (e.g., false and true, to stay within the CBOR simple values), if required.

Since null, false and true have single-byte representations, the replacement of NaN, −Infinity, and (positive) Infinity by these values can save bytes even if JSON compatibility is not a consideration.

Applications that need to preserve the information in a NaN (sign bit, quiet bit, payload) may want to replace null with an application-oriented representation of that information, or simply with a (left-aligned, truncating trailing zero bytes) byte string representing those bits:

```
float-with-nan-replacement = float / bytes
```

For JSON, the byte string can be base16- or base64-encoded, or it can be represented by an integer, preserving its left-aligned nature, or even as a (tagged) floating point value with a different exponent.

3.2.2. Implementation Considerations

All floating-point numbers, including zeros and infinities, are signed. A NaN also carries a sign bit. Each of the three formats binary16, binary32, and binary64 define a fixed assignment of bits in the representation towards the sign bit, an exponent, and a "significand" (which represents the mantissa, with details sometimes depending on the specific exponent value).

Format	Sign bit	Exponent	Significand
binary16	1	5	10
binary32	1	8	23
binary64	1	11	52

Table 1: Bit Allocation in Floating Point Formats

Infinite numbers are represented in each format choice with a sign bit, the highest available exponent value (all ones) and all-zero significand. NaN values are represented with a sign bit, the highest available exponent value (all ones) and a non-zero significand, which carries a leading quiet bit with the rest of the bits allocated to the NaN payload.

To qualify as a generic encoder or decoder, a CBOR library needs to implement as much of [IEEE754] support as reasonably possible on the platform it addresses. What is reasonably possible depends on:

* platform support for [IEEE754] numbers. If there is no such support, the generic decoder may need to resort to offering the interchanged value to the application, suitably tagged.

* If there is partial support, it may be harder to find a good solution. This is specifically a problem for platform support that works well in most cases, but exhibits some dark corners. E.g., the implementation may support a single NaN value consistently, but not preserve NaN information present in the NaN values.

Where an implementation needs to convert between different floating point formats, e.g., because not all formats are fully supported by the platform, or to implement Preferred Serialization (as needed for Common Deterministic Encoding [I-D.ietf-cbor-cde]) in an encoder, conversion of NaNs in these formats is best done by operating on the bit patterns of the [IEEE754] number in the following way:

* Expansion (towards a larger size format):

  - preserve the sign bit

  - expand the (all-ones) exponent to the larger (all-ones) exponent

- fill up the significand with zero bits on the right

* Contraction (towards a smaller size format):

  - preserve the sign bit

  - truncate the (all-ones) exponent to the smaller (all-ones) exponent

  - truncate the significand from the right; check if the removed bits were all zero.

If the contraction is optional, e.g., for Preferred Serialization, do not perform the contraction if the removed bits in the significand truncation are not all zero. If the contraction is required to fit into limited platform types (e.g., binary32 only), a failed truncation check indicates the loss of information and should be signaled to the application. We say a contraction "preserves the NaN information" if subsequent expansion to the original size format recreates the exact same NaN value.

Appendix B.1 gives additional detailed considerations for implementations that aspire to provide full support for NaNs, preserving NaN information.

3.3. Getting by without Platform Support for Floating Point Values

On a constrained platform, CPU instructions and library functions for IEEE 754 floating point values may not be available.

A partial implementation for constrained systems may want to restrict the application data models supported to those without floating point numbers.

However, IETF standards often use timestamps, which are efficiently supported by CBOR Tag 1. Tag 1 offers a choice between integer and floating point representations; only the latter are capable of resolutions of better than 1 second.

Appendix A shows how to support a useful subrange of Tag 1 time stamps that are encoded as floating point numbers for interchange, but processed as 32.32 bit fixed point numbers in a constrained implementation.

4. Other Floating Point Numbers

RFC 8949 [STD94] also defines tags 4 and 5 for a representation of decimal and binary floating point numbers that is not constrained by the types provided by IEEE 754. These tags are very flexible, but this flexibility comes with a choice of ways they could be integrated into a generic encoder. Because of this flexibility, tags 4 and 5 do not define a Preferred Serialization or a deterministic encoding.

Section 3.2 of [RFC9581] can use representations derived from the tags 4 and 5 to represent timestamps. Section 6.1 of [RFC9581] lists various other tags that can be used for representing numbers for advanced arithmetic, including rational numbers in fraction form (tag 30).

5. Tagged Arrays of Numbers

[RFC8746] defines tags for typed arrays, i.e., arrays of numbers that all are represented in the same way. The choices defined in the [RFC8746] are all based on traditional platform number representations (unsigned integers, signed integers, IEEE 754 floating point values) and even come in little-endian and big-endian variants, often removing the need to convert the numbers from an internal to an interchange form. As conversion for interchange is not envisioned, considerations for a preferred serialization are not applicable. As the recipient may need a conversion for ingestion of the arrays, some considerations from Section 3 may apply.

6. Security Considerations

The general security considerations for representing data in common data representation formats apply, e.g., those in Section 10 of RFC 8949 [STD94].

(TODO)

7. IANA Considerations

(TODO:

Add nan'' registration when that is ready)

8. References

8.1. Normative References

[BCP14] Best Current Practice 14, <https://www.rfc-editor.org/info/bcp14>.

At the time of writing, this BCP comprises the following:

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

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

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

[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>.

[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>.

[RFC9581] Bormann, C., Gamari, B., and H. Birkholz, "Concise Binary Object Representation (CBOR) Tags for Time, Duration, and Period", RFC 9581, DOI 10.17487/RFC9581, August 2024, <https://www.rfc-editor.org/rfc/rfc9581>.

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

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

8.2. Informative References

[ARM] Arm Limited, "Arm Architecture Reference Manual for A-profile architecture", April 2025, <https://developer.arm.com/documentation/ddi0487/latest/>.

[C23] International Organization for Standardization, "Information technology — Programming languages — C", ISO/IEC 9899:2024, October 2024, <https://www.iso.org/standard/82075.html>. This revision of the standard is widely known as C23. Technically equivalent specification text is available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf).

[Cplusplus20] International Organization for Standardization, "Programming languages - C++", Sixth Edition, ISO/IEC ISO/IEC JTC1 SC22 WG21 N 4860, March 2020, <https://isocpp.org/files/papers/N4860.pdf>.

[I-D.bormann-cbor-det] Bormann, C., "CBOR: On Deterministic Encoding and Representation", Work in Progress, Internet-Draft, draft-bormann-cbor-det-04, 21 January 2025, <https://datatracker.ietf.org/doc/html/draft-bormann-cbor-det-04>.

[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.mcnally-deterministic-cbor] McNally, W., Allen, C., Bormann, C., and L. Lundblade, "dCBOR: Deterministic CBOR", Work in Progress, Internet-Draft, draft-mcnally-deterministic-cbor-17, 11 February 2026, <https://datatracker.ietf.org/doc/html/draft-mcnally-deterministic-cbor-17>.

[USEFUL-CDDL] "Useful CDDL", n.d., <https://github.com/cbor-wg/cddl/wiki/Useful-CDDL#number-ranges>.

Appendix A. 32.32 Bit Fixed Point for Supporting Precision Tag 1 Timestamps

This appendix shows how to decode into and encode from a 32.32-bit internal fixed point representation of floating point Tag 1 time stamps, limited to a range from February 2004 to January 2106 inclusive. The fixed point representation is a 64-bit unsigned integer, the most significant half (fp >> 32) of which used as a 32-bit unsigned integer gives the timestamp rounded down to full seconds, and the least significant half of which (fp & 0xffffffff) contains a fractional part of the timestamp to a resolution of 2^-32 seconds (fractions of a nanosecond, more resolution than the approximately microseconds that Tag 1 actually can deliver in this range of timestamps).

The C functions given below return a 32.32-bit fixed point value as a 64-bit unsigned integer if the timestamp is covered in the above range, 0 otherwise. b64/b32 are the binary64/binary32 floating point values represented as an unsigned long (i.e., the CBOR argument value).

```
unsigned long timestamp_float64_to_fixed3232(unsigned long b64) {
  unsigned long exp = (b64 >> 52 & 0xfffUL) - 0x41d;
  if (exp >= 2)
    return 0;
  return ((b64 & 0xfffffffffffffUL)
          + 0x10000000000000UL) << (exp + 10);
}
```

Figure 1: Converting a float64 timestamp to fixed point 32.32

Because of preferred serialization, Tag 1 floating point timestamp values might arrive as a float32, so additional code is necessary for handling this rare case:

```
unsigned long timestamp_float32_to_fixed3232(unsigned long b32) {
  unsigned long exp = (b32 >> 23 & 0x1ffUL) - 0x9d;
  if (exp >= 2)
    return 0;
  return ((b32 & 0x7fffffUL) + 0x800000UL) << (exp + 39);
}
```

Figure 2: Converting a float32 timestamp to fixed point 32.32

There is no value in the timestamp range supported by this simple code that would fit into a float16.

Appendix B. Implementers' Checklists for Floating Point Values

This check list employs [BCP14] keywords to indicate interoperability requirements on implementations.

The following considerations apply to encoding (emitting) floating point values in a generic encoder:

* The length of the argument is encoded in the lower 5 bits of the first byte ("ai"), which indicates half precision (binary16, ai = 0x19), single precision (binary32, ai = 0x1a) and double precision (binary64, ai = 0x1b).

  For preferred serialization: if multiple of these encodings preserve the precision of the value to be encoded, only the shortest form of these MUST be emitted. That implies that encoders MUST support half-precision and (if there is support for more than half precision on the platform) single-precision floating point. Positive and negative infinity and zero MUST be represented in half-precision floating point.

* NaNs MUST be supported, for all values of NaN information allowed in [IEEE754].

  As with all floating point numbers, NaNs with payloads MUST be contracted to the shortest of double, single or half precision that preserves the NaN information.

  The reduction is performed by removing the rightmost N bits of the payload, where N is the difference in the number of bits in the significand (mantissa) between the original format and the reduced format. The reduction is performed only (preserves the value only) if all the rightmost bits removed are zero. (This will always reduce a double or single quiet NaN with an otherwise zero NaN payload, which is typically what is returned from an operation such as 0.0/0.0, to a half-precision quiet NaN encoded as 0xf9 7e00.)

The following considerations apply to decoding (ingesting) floating point values in a generic decoder that supports IEEE 754 floating-point numbers:

* Half-precision values MUST be accepted.

* Double- and single-precision values SHOULD be accepted; leaving these out is only foreseen for decoders that need to work in exceptionally constrained environments.

* If double-precision values are accepted, single-precision values MUST be accepted.

* NaNs, MUST be accepted, preserving the NaN information for use of the application.

B.1. NaN Payloads

The basic data model of CBOR directly supports IEEE-754 data item of the forms binary16, binary32, and binary64. These have 10, 23, and 52 bits in the space provided for encoding the significand (see Table 1). For a NaN, the first of these bits is used to indicate whether the NaN is signalling (0) or quiet (1). The up to 51 bits in the rest of the significand are called the "NAN payload".

The payload's original purpose is diagnostic information to explain why a NaN was generated by a local computation. There is no standard for the contents of a NaN payload.

CBOR allows NaNs with non-zero payloads to be encoded. (Due to the way infinite numbers are encoded in [IEEE754], zero-payload NaN always must be quiet NaNs.)

As a result, if a protocol design does not use NaNs with non-zero payloads and is using preferred serialization then NaN must be encoded as a half-precision with the quiet bit set and the payload set as 0, specifically 0xF97E00. If a design does not use NaNs with non-zero payloads and preferred serialization is not used, then the single and double precision quiet NaNs, 0xFA7FC00000 and 0xFB7FF0000000000000, may also be used.

B.1.1. Working with NaNs

NaN payloads have been in the IEEE-754 standard since 2008, but programming environments often still do not provide facilities (e.g., APIs) to make full use of them. In C there is the isnan() API to check if a value is a NaN, but there are only implementation-defined APIs to construct or access the NaN payload [C23]. For constructing a NaN with a payload, C for instance offers functions specific to a floating point type, such as nanf() for single precision (binary32) and nan() for double precision (binary64), or their analogues in the strtof() and strtol() functions. Section 7.24.1.5 and its Footnote 341 helpfully explain:

> [...] the meaning of the n-char sequence is implementation-defined. [...] An implementation can use the n-char sequence to determine extra information to be represented in the NaN's significand.

While Section 9.7 of [IEEE754] now defines abstract APIs for creating and accessing NaN values (getPayload/setPayload), these definitions are partially implementation-defined and are vague enough that realizations of them have not made it into [C23].

The typical way to work with a NaN payload in C instead is to reinterpret the floating-point value as an unsigned integer and then use shifts and masks to unpack the IEEE-754 representation.

The floating point conversion (narrowing/widening) instructions of the most widely used CPU architectures cover NaNs in a comparable way as they convert finite numbers: They narrow by removing the right-most bits of the payload, and they widen by adding zero bits to the right. (E.g., for ARM A-profile Architecture [ARM]: See Section J1.3.3.193 FPConvertNaN, page 14208 at the time of writing.)

B.1.2. NaN Implementation Details

This section is primarily for CBOR library implementors.

CBOR attempts to limit the MUSTs about CBOR implementations in order to allow its use in a large variety of constrained use cases. For example, support for integers is not required because a protocol might need only strings. Similarly, there is no MUST that requires support of NaN and NaNs with non-zero payloads, but the recommendation here is that any generic CBOR library that supports floating-point support NaNs, preferably also with non-zero NaN payloads.

In most environments, there is little extra work to do to support NaN without payloads if floating-point is supported. NaNs will usually flow through as any other floating-point value.

Generic CBOR libraries are expected to support preferred serialization of floating-point including NaNs. For NaNs with zero payloads, this requires reducing to a half-precision NaN without a payload. This requires a few explicit extra lines of code. See the sample half-precision implementation in Appendix D of RFC 8949.

The implementation of preferred serialization of NaN payloads needs a few more additional lines. As with preferred serialization, NaN payloads must be reduced but only if they can be reduced without the loss of any non-zero payload bits. Programming platform provided floating-point hardware and software may or may not do this correctly for double to single conversion. The sample half-precision implementation in Appendix D of RFC 8949 only supports NaNs without payloads.

A double precision NaN payload contains 51 bits, a single 22 bits and a half 9 bits, in each case all but the first bit of the significand. A double precision NaN can be reduced to a single precision NaN only if the right-most 29 payload bits are zero. A single precision NaN can be reduced to a half precision NaN only if the right-most 13 payload bits are zero. A double NaN can be reduced to a half precision NaN only if the right-most 42 payload bits are zero. Note that the exponent is always all-ones for NaN, so this is simpler than the equivalent contraction of regular, non-NAN, floating-point values.

To implement the above, most CBOR libraries will have to reinterpret the floating point value as an unsigned integer and use shifts and masks, based in the internal representation defined in [IEEE754].

Testing on some CPUs has shown them to do this correctly for conversion between single and double. However, it may not be very useful to rely on platform libraries for the following reasons. First, they may provide no support at all for half-precision and half-precision is required for preferred serialization. Second, NaN payloads are a relatively recent and very specialist feature that is not usually used in interchange.

If platform implementation is relied upon, NaN payload reduction should be tested on each platform. Open source libraries intended to run on multiple platforms may be better off not relying on the platform.

B.1.3. NaN Tests Examples

The IEEE-754 numbers are given as a 64-bit (binary64) or 32-bit (binary32) unsigned integer in hex to show the bits that make up the floating-point value. All of the following are NaNs.

IEEE-754 Number	CBOR Preferred Serialization	Comment
0x7ff8000000000000	0xf97e00	qNaN contracted from double to half
0x7ff8000000000001	0xfb7ff8000000000001	Can’t be contracted because of bit set in right-side part of payload
0x7ffffc0000000000	0xf97fff	10-bit payload that can be contracted to half
0x7ff80000000003ff	0xfb7ff80000000003ff	right-justified payload can’t be contracted
0x7fffffffe0000000	0xfa7fffffff	23-bit payload that reduces to single
0x7fffffffff0000000	0xfb7fffffffff0000000	24-bit payload that can’t be contracted
0x7fffffffffffffff	0xfb7fffffffffffffff	All payload bits set, can’t be contracted
0x7fc00000	0xf97e00	qNaN contracted from single to half
0x7fffe000	0xf97fff	single 10-bit payload that can be contracted
0x7fbff000	0xfa7fbff000	single payload that can’t be contracted to 10 bits

Table 2: Examples for Preferred Serialization of NaN values

List of Figures

List of Tables

Acknowledgments

Contributors

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

Laurence wrote much of the initial text about NaN processing.

Author’s Address

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