Abstract

This specification addresses the challenges of using JavaScript Object Notation (JSON) with constrained devices by providing a standard set of mapping rules to Concise Binary Object Representation (CBOR) that preserve all semantic information, such that the original JSON string can be identically re-created. JSON Constrained Notation can also be used by devices as a native data format, which can then be represented as JSON when necessary for diagnostics, compatibility, and ease of integration with higher-level systems.

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 19, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction ................................................. 2
   1.1 Terminology ........................................... 4
2. CBOR Encoding ................................................. 4
   2.1 Structured Types ....................................... 4
   2.2 Primitive Types ....................................... 5
      2.2.1 Boolean and Null ................................... 5
      2.2.2 Numbers .......................................... 5
      2.2.3 Strings .......................................... 5
3. Reference Sets ............................................... 6
4. Canonical Form ............................................... 7
   4.1 Formatting-only Whitespace ............................... 7
   4.2 String Escapes ......................................... 9
5. Constrained API .............................................. 10
6. Examples ................................................... 10
   6.1 JSON .................................................. 10
      6.1.1 Input ............................................. 10
      6.1.2 Optimized JSCN Encoding ........................... 11
      6.1.3 Un-optimized JSCN Encoding ........................ 13
   6.2 JSON Web Token ......................................... 15
7. IANA Considerations .......................................... 16
   7.1 CBOR Tags ............................................... 16
   7.2 JSCN Reference Sets Registry ........................... 17
8. Security Considerations ...................................... 17
9. References .................................................. 17
   9.1 Normative References ................................... 17
   9.2 Informative References ................................ 18
   9.3 URIs .................................................. 18
Appendix A. Acknowledgements .................................. 18
Authors’ Addresses ............................................. 19

1. Introduction

Although JavaScript Object Notation (JSON) [RFC7159] has been widely adopted in traditional networking and software environments, its use in embedded and constrained environments has been more limited because of the minimal storage and network capacities inherent in low-cost and low-power devices (see [RFC7228]).

This specification addresses the challenges of using JSON with constrained devices by defining a set of mapping rules to Concise Binary Object Representation (CBOR) [RFC7049] that preserve all
semantic information, such that the original JSON string can be
identically re-created. JSON Constrained Notation (JSCN) can be used
directly by devices as a native data format, which can be represented
as JSON when necessary for diagnostics, compatibility, and ease of
integration with higher-level systems.

A primary goal of JSCN is to enable all JSON Object Signing and
Encryption (JOSE) standards ([RFC7515], [RFC7516], [RFC7517],
[RFC7518], [RFC7519]) to be used unmodified in constrained
environments. One result is that OpenID Connect [1] (which utilizes
JSON Web Tokens [RFC7519]) can more easily be adopted as an identity
management solution for the Internet of Things.

JSCN is designed to leverage, not replace, CBOR. Instead, JSCN
specifies rules for re-coding JSON structures by mapping them to
their CBOR parallels whenever possible, and then increasing the
efficiency through introspection and replacement of well-known
strings with compact references.

All transcoding software MUST operate on a UTF-8 JSON string whenever
complete round-trip compatibility to and from JSON is required,
including mapping any contained non-structural whitespace (such as
with JWTs for signature validation). If a transcoder is only
operating with an already parsed JSON value (the result of
"JSON.parse()" in JavaScript for instance), the round-trip can only
guarantee semantic compatibility of the values as represented in that
parsed context (only the JavaScript object will always match).

A significant reduction in space is also provided in JSCN when the
device and application contexts can make use of built-in or shared
UTF-8 string references. These references provide a mapping of
common JSON string values to an integer that used to replace the
string in the resulting CBOR during re-coding. JSON string values
are also introspected for data that has a more compact CBOR type
(such as base64url and hexadecimal encoding).

The use of this specification can ensure that a UTF-8 JSON string
before and after re-coding will be byte-for-byte identical across
implementations, whereas the CBOR encoding is not designed to have
this property and MAY vary based on implementation choices and
reference sets available. There are basic API rules defined for
constrained software such that directly accessing the CBOR data
values will always provide a uniform view to an application across
variations in the underlying CBOR representation.
1.1. Terminology

Many terms used in this document are defined in the specifications for JSON [RFC7159] and CBOR [RFC7049]. This specification defines the following additional terms:

- **Constrained JSON Tag**
  - The CBOR tag registered in this specification to indicate an array that contains JSON data encoded as CBOR according to this specification.

- **Reference**
  - A pointer within JSCN data that refers to a well-known UTF-8 string by using a CBOR byte string of length one, where the byte value is the lookup identifier for the Reference.

- **Reference Set**
  - A CBOR array of UTF-8 strings that are used to replace any Reference within any JSCN data, where the Reference identifier is the array offset to the replacement string and the first position in the array identifies the Reference Set.

- **Canonical Hints**
  - A CBOR array of integers that indicate positional offsets for JSON string escape sequences or structural formatting whitespace strings (, \n, \r, and \t) such that when any CBOR encoded data is stringified into JSON it can also optionally be corrected to exactly match the original JSON string.

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 [RFC2119].

2. CBOR Encoding

JSCN encodes JSON data types to CBOR data types as described in the following sections.

2.1. Structured Types

JSON defines two structured types: arrays and objects. These are serialized to CBOR major type 4 (array) and type 5 (map),
respectively. Ordering of key/value pairs in JSON objects and CBOR maps MUST be preserved.

2.2. Primitive Types

2.2.1. Boolean and Null

The JSON literal names "false", "true", and "null" are serialized to the CBOR major type 7 simple values 20, 21, and 22 respectively.

2.2.2. Numbers

A JSCN encoder attempts to encode a JSON number as a CBOR unsigned integer (type 0), negative integer (type 1), or float (type 7) and then test for compatibility by round-tripping the CBOR data item back to a JSON number. If the resulting JSON number is not equivalent to the input number, the encoder MUST instead encode it as a CBOR Bigfloat (tag 5).

The JSON exponent value (if any) is encoded as a CBOR exponent (tag 4). If the contained "e" symbol is upper case in JSON, the "Upper Case Modifier" tag defined below MUST be included.

2.2.3. Strings

A JSON string is normally encoded as an un-escaped CBOR UTF-8 string (type 3), i.e., as a series of UTF-8 [RFC3629] characters (e.g., the word "one" is encoded as "6F6E65") without any backspace escaping for control or unicode characters.

2.2.3.1. Base64 / Base16 Encoded

A JSCN encoder MUST round-trip test all JSON strings for possible encodings (base64url, base64, and hexadecimal) by attempting to decode and re-encode them. If identical byte strings result, the decoded value is tagged in CBOR with the encoding format (tags 21, 22, and 23). For hexadecimal, the "Upper Case Modifier" tag defined below MUST be included if the hexadecimal letters A-F are upper case in the original JSON string.

A JSCN encoder MUST perform introspection on the resulting decoded byte string to determine if it begins with a JSON structure byte of '{' or '['. The encoder SHOULD then round-trip test the string as a possible JSON object or array so that it can encode the string more efficiently into a CBOR data item instead of a byte string (this pattern is common in the JOSE specification).
3. Reference Sets

The Constrained JSON Tag is followed by an array whose second item identifies the Reference Set used in the data. This is either a Reference Set identifier or an array that defines an inline Reference Set.

A Reference Set identifier is a unique integer that maps to a Reference Set known to applications using the set. Public, well-known reference sets can be registered as described in the IANA Considerations section of this document.

The Reference Set definition is encoded as a JSCN array, where the first value is the Reference Set identifier followed by all of the UTF-8 string keys. A key’s position in the array is the byte value with which it is replaced.

Any Reference Set can include another Reference Set by encoding the second set’s identifier in the JSCN array that defines the first Reference Set. Any byte strings in the definition array are then replaced with the key from the references contained in the second Reference Set.

JSON UTF-8 strings representing keys or values are first checked against all active references (if any) for possible replacement. A replacement is always a CBOR byte string (type 2) of length 1, where the single byte represents the index value of the key in the references array from 1-255. Value 0 and byte lengths greater than 1 are reserved for future use.

When a JSCN decoder generates JSON values from CBOR and it encounters a CBOR byte string (type 2), single byte value MUST match the array offset of the active references to be used as the replacement for that byte string.

The following is the encoded form of a Reference Set as defined by the JSON array of `[1,"map","value","array","one","two","three","bool ","neg","simple","ints"]`:
4. Canonical Form

This specification directly supports use-cases such as JSON Web Tokens ([RFC7518]) where the canonical form of UTF-8 JSON strings always needs to be available for validation. This is accomplished by optionally including any additional information to reproduce the exact UTF-8 string as an array of Canonical Hints included with the Constrained JSON Tag.

These hints are not typically necessary as most machine-generated JSON does not include any extra insignificant bytes by default, even when included they do not need to be processed unless the original canonical form is requested. When required, these additional hints also take a highly constrained form and are independently additive to the contained CBOR data values such that those values remain uniform to any constrained application.

4.1. Formatting-only Whitespace

When a Constrained JSON tag is present and the first item in the tagged array is a CBOR structure (map or array), a third optional item in the tagged array is a set of canonical whitespace hints for
any non-structural whitespace characters contained in the original
UTF-8 representation of the JSON object or array.

- Whitespace hints are contained in an array of integers that
  indicate offsets of the locations of whitespace characters in an
  original JSON string, and lookup values identifying which
  whitespace characters were there.

- Each offset integer is relative to the position of the previous
  offset such that all integers are of small values.

- A negative integer offset indicates a single ASCII space character
  (0x20) at the offset of the positive value of that integer.

- An unsigned integer offset is followed by another integer, where
  unsigned values (0-23) indicate a whitespace string in the pre-
  defined lookup table, and negative values specify the number of
  space characters (0x20) to repeat.

- When re-inserting whitespace characters to a JSON string, the
  array MUST be applied sequentially so that each new offset matches
  the original JSON string position.

The following 24 whitespace character hexadecimal sequences are used
as the shared reference lookup table by row (0-23) when processing
whitespace hints. This table is constructed to minimize the number
of references commonly required while also allowing any possible
whitespace character sequences to be identified.
4.2. String Escapes

JSON string values MAY contain escaped characters (as defined in Section 7 of [RFC7159]) that become un-escaped in the process of re-coding them into a CBOR UTF-8 string. When the canonical form is being preserved and any escaped characters are detected in the process of converting them from JSON to CBOR, those string values MUST be individually tagged as Constrained JSON where the first element in the tagged array is the CBOR UTF-8 string value and the second value is an array of positional integers similar to the whitespace hints.

When the position is an unsigned integer it indicates the UTF-8 character at that position is to be escaped with the "\uXXXX" form with lower-case hexadecimal characters. When it is a negative integer it indicates that it is to be escaped with the "\X" form and MUST be in the set of JSON escaped control characters.

When the original escaping in the "\uXXXX" form was with upper case hexadecimal characters the entire array MUST be tagged with Upper Case Modifier. In the unlikely case that the original escaping contained mixed-case hexadecimal, then the positional integer will instead itself be an array of length two with the position being the
first element and a 4-byte UTF-8 string of the mixed-case hexadecimal value being the second element.

5. Constrained API

In order to ease the use of JSCN in constrained environments, an implementation SHOULD make data values available both as native CBOR types and as JSON strings; this enables a constrained application to choose either format regardless of how the data is represented in CBOR.

For example, when the original JSON string value is encoded as a CBOR base64url tag plus byte string, a constrained application accessing the value as a string MUST receive the base64url encoded value and not the decoded byte value. If the constrained application instead accesses the value as a byte array it MUST get the decoded value if available.

The representation of the value in CBOR SHOULD NOT alter behavior of the application, a string value encoded as tag plus byte array SHOULD NOT be used as an indication that it is a binary value and only the application can make this determination based on external context.

6. Examples

6.1. JSON

6.1.1. Input

Consider the following JSON as input to a JSCN encoder.
6.1.2. Optimized JSCN Encoding

An optimized encoding would remove whitespace and use a Reference Set. Here the references would be:

"[1,"map","value","array","one","two","three","bool","neg","simple","ints"]"

The resulting JSCN encoding is 90 bytes compared to 318 bytes for the JSON input.
D4  # tag(20)
82  # array(2)
A6  # map(6)
  41  # bytes(1)
   01  # "\x01"
  41  # bytes(1)
   02  # "\x02"
  41  # bytes(1)
   03  # "\x03"
  84  # array(4)
   41  # bytes(1)
    04  # "\x04"
   41  # bytes(1)
    05  # "\x05"
   41  # bytes(1)
    06  # "\x06"
  18 2A  # unsigned(42)
   41  # bytes(1)
    07  # "\a"
   F5  # primitive(21)
   41  # bytes(1)
    08  # "\b"
  38 29  # negative(41)
   41  # bytes(1)
    09  # "\t"
  83  # array(3)
   F4  # primitive(20)
   F6  # primitive(22)
  60  # text(0)
    # ""
   41  # bytes(1)
  0A  # "\n"
  8C  # array(12)
  00  # unsigned(0)
  01  # unsigned(1)
  17  # unsigned(23)
  18 18  # unsigned(24)
  19 00FF  # unsigned(255)
  19 0100  # unsigned(256)
  19 FFFF  # unsigned(65535)
  1A 00010000  # unsigned(65536)
  1B 00000000FFFFFF  # unsigned(4294967295)
  1B 0000001000000000  # unsigned(4294967296)
  1B 0001000000000000  # unsigned(28147976710656)
  3B 0000FFFFFFF  # negative(28147976710655)
  01  # unsigned(1)
6.1.3. Un-optimized JSCN Encoding

An un-optimized encoding would not use a Reference Set and would preserve whitespace. The un-optimized encoding would reduce the data from the 318 bytes (JSON) to 187 bytes (JSCN).

```
D4  # tag(20)
83  # array(3)
A6  # map(6)
   # text(3)
   6D6170 # "map"
   65  # text(5)
   76616C7565 # "value"
   65  # text(5)
   6172766179 # "array"
84  # array(4)
   # text(3)
   6F6E65 # "one"
   63  # text(3)
   746F626565F5 # primitive(21)
   63  # text(3)
   6E6567 # "neg"
   38  # negative(41)
   29  # text(6)
   73696D706C65 # "simple"
83  # array(3)
   F4  # primitive(20)
   F6  # primitive(22)
   60  # text(0)
       # ""
64  # text(4)
   696E7473 # "ints"
8C  # array(12)
   00  # unsigned(0)
   01  # unsigned(1)
   17  # unsigned(23)
   18 18  # unsigned(24)
   19 00FF # unsigned(255)
   19 0100 # unsigned(256)
   19 FFFF # unsigned(65535)
   1A 00010000 # unsigned(65536)
   1B 000000000000000000FF # unsigned(4294967295)
```
1B 0000000100000000 # unsigned(4294967296)
1B 0001000000000000 # unsigned(281474976710656)
3B 0000FFFFFFFFFFFFF # negative(281474976710655)
00 # unsigned(0)
98 40 # array(64)
01 # unsigned(1)
01 # unsigned(1)
26 # negative(6)
08 # unsigned(8)
01 # unsigned(1)
28 # negative(8)
01 # unsigned(1)
02 # unsigned(2)
06 # unsigned(6)
02 # unsigned(2)
06 # unsigned(6)
02 # unsigned(2)
08 # unsigned(8)
02 # unsigned(2)
02 # unsigned(2)
01 # unsigned(1)
02 # unsigned(2)
01 # unsigned(1)
27 # negative(7)
05 # unsigned(5)
01 # unsigned(1)
26 # negative(6)
04 # unsigned(4)
01 # unsigned(1)
29 # negative(9)
01 # unsigned(1)
02 # unsigned(2)
06 # unsigned(6)
02 # unsigned(2)
05 # unsigned(5)
02 # unsigned(2)
02 # unsigned(2)
01 # unsigned(1)
02 # unsigned(2)
01 # unsigned(1)
27 # negative(7)
01 # unsigned(1)
02 # unsigned(2)
02 # unsigned(2)
02 # unsigned(2)
02 # unsigned(2)
03 # unsigned(3)
6.2. JSON Web Token

Consider the following JSON Web Token [RFC7519], which natively is 149 bytes (line endings are not significant):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDCefxjoYZgeFONFh7HgQ
```

In a JSON encoding, the JWT would be 191 bytes (line endings are not significant):

```
{"protected": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9", "payload": "eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9", "signature": "TJVA95OrM7E2cBab30RMHrHDCefxjoYZgeFONFh7HgQ"}
```

Using a Reference Set of 

```
[1,"payload","signature","protected","alg","HS256","sub","name","admin"]
```

the JSCN encoding would be 80 bytes.
7. IANA Considerations

7.1. CBOR Tags

The IANA is requested to assign the following tags from the "CBOR Tags" registry defined in RFC 7049 [RFC7049]:
o Assign the tag "Constrained JSON" in the 1 to 23 value range (one byte in length when encoded).

o Assign the tag "Upper Case Modifier" in the 24 to 255 value range (two bytes in length when encoded).

The tags to be assigned are described below.

<table>
<thead>
<tr>
<th>Tag</th>
<th>20 (Constrained JSON)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Item</td>
<td>array</td>
</tr>
<tr>
<td>Semantics</td>
<td>The first value in the array is a constrained JSON data item encoded using JSCN, optionally followed by an integer or array identifying any embedded references, and then an optional array of canonical hints (if any).</td>
</tr>
<tr>
<td>Reference</td>
<td><a href="http://quartzjer.github.io/JSCN">http://quartzjer.github.io/JSCN</a></td>
</tr>
<tr>
<td>Contact</td>
<td>Jeremie Miller <a href="mailto:jeremie.miller@gmail.com">jeremie.miller@gmail.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>31 (Upper Case Modifier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Item</td>
<td>multiple</td>
</tr>
<tr>
<td>Semantics</td>
<td>Indicates that the data item following contains values where the upper case is semantically important when interpreted in a UTF-8 string context.</td>
</tr>
<tr>
<td>Reference</td>
<td><a href="http://quartzjer.github.io/JSCN">http://quartzjer.github.io/JSCN</a></td>
</tr>
<tr>
<td>Contact</td>
<td>Jeremie Miller <a href="mailto:jeremie.miller@gmail.com">jeremie.miller@gmail.com</a></td>
</tr>
</tbody>
</table>

7.2. JSCN Reference Sets Registry

A future version of this document will request creation of a registry for JSCN Reference Sets and provide initial registrations for the existing JOSE JWE, JWS, and JWA RFCs.

8. Security Considerations

TODO

9. References

9.1. Normative References

9.2. Informative References


9.3. URIs


Appendix A. Acknowledgements

Thanks to Carsten Bormann and David Waite for their comments.
Authors’ Addresses

Jeremie Miller
Email: jeremie@jabber.org

Peter Saint-Andre
Filament
Email: peter@filament.com