            Binary Encodings for JavaScript Object Notation: JSON-B, JSON-C, JSON-D
                       draft-hallambaker-jsonbcd-10

Abstract

   Three binary encodings for JavaScript Object Notation (JSON) are
   presented. JSON-B (Binary) is a strict superset of the JSON encoding
   that permits efficient binary encoding of intrinsic JavaScript data
   types. JSON-C (Compact) is a strict superset of JSON-B that supports
   compact representation of repeated data strings with short numeric
   codes. JSON-D (Data) supports additional binary data types for
   integer and floating-point representations for use in scientific
   applications where conversion between binary and decimal
   representations would cause a loss of precision.

   This document is also available online at
   http://prismproof.org/Documents/draft-hallambaker-jsonbcd.html [1] .

Status of This Memo

Copyright Notice

Table of Contents

[1](#). Introduction

   JavaScript Object Notation (JSON) is a simple text encoding for the
   JavaScript Data model that has found wide application beyond its
   original field of use. In particular JSON has rapidly become a
   preferred encoding for Web Services.

   JSON encoding supports just four fundamental data types (integer,
   floating point, string and boolean), arrays and objects which consist
   of a list of tag-value pairs.

   Although the JSON encoding is sufficient for many purposes it is not
   always efficient. In particular there is no efficient representation
   for blocks of binary data. Use of base64 encoding increases data
   volume by 33%. This overhead increases exponentially in applications

where nested binary encodings are required making use of JSON
encoding unsatisfactory in cryptographic applications where nested
binary structures are frequently required.

Another source of inefficiency in JSON encoding is the repeated
occurrence of object tags. A JSON encoding containing an array of a
hundred objects such as {"first":1,"second":2} will contain a hundred
occurrences of the string "first" (seven bytes) and a hundred
occurrences of the string "second" (eight bytes). Using two byte
code sequences in place of strings allows a saving of 11 bytes per
object without loss of information, a saving of 50%.

A third objection to the use of JSON encoding is that floating point
numbers can only be represented in decimal form and this necessarily
involves a loss of precision when converting between binary and
decimal representations. While such issues are rarely important in
network applications they can be critical in scientific applications.
It is not acceptable for saving and restoring a data set to change
the result of a calculation.

1.1. Objectives

The following were identified as core objectives for a binary JSON
encoding:

o Easy to convert existing encoders and decoders to add binary
   support

o Efficient encoding of binary data

o Ability to convert from JSON to binary encoding in a streaming
   mode (i.e. without reading the entire binary data block before
   beginning encoding.

o Lossless encoding of JavaScript data types

o The ability to support JSON tag compression and extended data
   types are considered desirable but not essential for typical
   network applications.

Three binary encodings are defined:

JSON-B (Binary) Encodes JSON data in binary. Only the JavaScript
   data model is supported (i.e. atomic types are integers, double or
   string). Integers may be 8, 16, 32 or 64 bits either signed or
   unsigned. Floating points are IEEE 754 binary64 format [IEEE754]
   . Supports chunked encoding for binary and UTF-8 string types.

JSON-C (Compact) As JSON-B but with support for representing JSON
   tags in numeric code form (16 bit code space). This is done for
   both compact encoding and to allow simplification of encoders/
   decoders in constrained environments. Codes may be defined inline
   or by reference to a known dictionary of codes referenced via a
   digest value.

JSON-D (Data) As JSON-C but with support for representing additional
   data types without loss of precision. In particular other IEEE
   754 floating point formats, both binary and decimal and Intel's 80
   bit floating point, plus 128 bit integers and bignum integers.

Each encoding is a proper superset of JSON, JSON-C is a proper
superset of JSON-B and JSON-D is a proper superset of JSON-C. Thus a
single decoder MAY be used for all three new encodings and for JSON.
Figure 1 shows these relationships graphically:

[[This figure is not viewable in this format. The figure is
available at http://prismproof.org/Documents/draft-hallambaker-
jsonbcd.html [2].]]

Encoding Relationships.

2. Definitions

This section presents the related specifications and standard, the
terms that are used as terms of art within the documents and the
terms used as requirements language.

2.1. Requirements Language

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 [RFC2119].

2.2. Defined Terms

The terms of art used in this document are described in the Mesh
Architecture Guide [draft-hallambaker-mesh-architecture] .

2.3. Related Specifications

The JSON-B, JSON-C and JSON-D encodings are all based on the JSON
grammar [RFC7159] . IEEE 754 Floating Point Standard is used for
encoding floating point numbers [IEEE754] ,

2.4. Terminology

No new terms of art are defined

3. Extended JSON Grammar

The JSON-B, JSON-C and JSON-D encodings are all based on the JSON
grammar [RFC7159] using the same syntactic structure but different
lexical encodings.

JSON-B0 and JSON-C0 replace the JSON lexical encodings for strings
and numbers with binary encodings. JSON-B1 and JSON-C1 allow either
lexical encoding to be used. Thus any valid JSON encoding is a valid
JSON-B1 or JSON-C1 encoding.

The grammar of JSON-B, JSON-C and JSON-D is a superset of the JSON
grammar. The following productions are added to the grammar:

x-value Binary encodings for data values. As the binary value
   encodings are all self delimiting

x-member An object member where the value is specified as an X-value
   and thus does not require a value-separator.

b-value Binary data encodings defined in JSON-B.

b-string Defined length string encoding defined in JSON-B.

c-def Tag code definition defined in JSON-C. These may only appear
   before the beginning of an Object or Array and before any
   preceding white space.

c-tag Tag code value defined in JSON-C.

d-value Additional binary data encodings defined in JSON-D for use
   in scientific data applications.

The JSON grammar is modified to permit the use of x-value productions
in place of (value value-separator) :

JSON-text = (object / array)

Figure 1

Figure 2

```
    object = *cdef begin-object [
```

Figure 3

```
         *( member value-separator | x-member )
```

Figure 4

```
         (member | x-member) ] end-object
```

Figure 5

Figure 6

```
  member = tag value
```

Figure 7

```
  x-member = tag x-value
```

Figure 8

Figure 9

```
  tag = string name-separator | b-string | c-tag
```

Figure 10

Figure 11

```
  array = *cdef begin-array [   *( value value-separator | x-value )
```

Figure 12

```
(value | x-value) ] end-array
```

Figure 13

Figure 14

```
x-value = b-value / d-value
```

Figure 15

Figure 16

```
value = false / null / true / object / array / number / string
```

Figure 17

Figure 18

```
name-separator  = ws %x3A ws  ; : colon
```

Figure 19

```
value-separator = ws %x2C ws  ; , comma
```

Figure 20

The following lexical values are unchanged:

Figure 21

```
begin-array     = ws %x5B ws  ; [ left square bracket
```

Figure 22

```
begin-object    = ws %x7B ws  ; { left curly bracket
```

Figure 23

```
end-array       = ws %x5D ws  ; ] right square bracket
```

Figure 24

```
end-object      = ws %x7D ws  ; } right curly bracket
```

Figure 25

Figure 26

```
ws = *( %x20 %x09 %x0A  %x0D )
```

Figure 27

Figure 28

```
false = %x66.61.6c.73.65   ; false
```

Figure 29

```
null  = %x6e.75.6c.6c      ; null
```

Figure 30

```
true  = %x74.72.75.65      ; true
```

Figure 31

The productions number and string are defined as before:

```
number = [ minus ] int [ frac ] [ exp ]
```

Figure 32

```
decimal-point = %x2E        ; .
```

                              Figure 33

```
digit1-9 = %x31-39          ; 1-9
```

                              Figure 34

```
e = %x65 / %x45             ; e E
```

                              Figure 35

```
exp = e [ minus / plus ] 1*DIGIT
```

                              Figure 36

```
frac = decimal-point 1*DIGIT
```

                              Figure 37

```
int = zero / ( digit1-9 *DIGIT )
```

                              Figure 38

```
minus = %x2D                ; -
```

                              Figure 39

```
plus = %x2B                 ; +
```

                              Figure 40

```
zero = %x30                 ; 0
```

                              Figure 41

                              Figure 42

    string = quotation-mark *char quotation-mark

                            Figure 43

    char = unescaped /

                            Figure 44

    escape (%x22 / %x5C / %x2F / %x62 / %x66 /

                            Figure 45

    %x6E / %x72 / %x74 / %x75 4HEXDIG)

                            Figure 46

                            Figure 47

    escape = %x5C   ; \

                            Figure 48

    quotation-mark = %x22   ; "

                            Figure 49

    unescaped = %x20-21 / %x23-5B / %x5D-10FFFF

                            Figure 50

4. JSON-B

    The JSON-B encoding defines the b-value and b-string productions:

    b-value = b-atom | b-string | b-data | b-integer |

                            Figure 51

b-float

Figure 52

Figure 53

b-string = *(string-chunk) string-term

Figure 54

b-data = *(data-chunk) data-last

Figure 55

Figure 56

b-integer = p-int8 | p-int16 | p-int32 | p-int64 | p-bignum16 |

Figure 57

n-int8 | n-int16 | n-int32 | n-int64 | n-bignum16

Figure 58

Figure 59

b-float = binary64

Figure 60

The lexical encodings of the productions are defined in the following
tables where the column 'tag' specifies the byte code that begins the
production, 'Fixed' specifies the number of data bytes that follow
and 'Length' specifies the number of bytes used to define the length
of a variable length field following the data bytes:

Production	Tag	Fixed	Length	Data Description
string-term	x80	-	1	Terminal String 8 bit length
string-term	x81	-	2	Terminal String 16 bit length
string-term	x82	-	4	Terminal String 32 bit length
string-term	x83	-	8	Terminal String 64 bit length
string-chunk	x84	-	1	Terminal String 8 bit length
string-chunk	x85	-	2	Terminal String 16 bit length
string-chunk	x86	-	4	Terminal String 32 bit length
string-chunk	x87	-	8	Terminal String 64 bit length
data-term	x88	-	1	Terminal String 8 bit length
data-term	x89	-	2	Terminal String 16 bit length
data-term	x8A	-	4	Terminal String 32 bit length
data-term	x8B	-	8	Terminal String 64 bit length
data-term	X8C	-	1	Terminal String 8 bit length
data-term	x8D	-	2	Terminal String 16 bit length
data-term	x8E	-	4	Terminal String 32 bit length
data-term	x8F	-	8	Terminal String 64 bit length

Table 1

Table 1: Codes for String and Data items

```
+------------+-----+-------+--------+-----------------------------+
| Production | Tag | Fixed | Length | Data Description            |
+------------+-----+-------+--------+-----------------------------+
| p-int8     | xA0 | 1     | -      | Positive 8 bit Integer      |
| p-int16    | xa1 | 2     | -      | Positive 16 bit Integer     |
| p-int32    | xa2 | 4     | -      | Positive 32 bit Integer     |
| p-int64    | xa3 | 8     | -      | Positive 64 bit Integer     |
| p-bignum16 | Xa7 | -     | 2      | Positive Bignum             |
| n-int8     | xA8 | 1     | -      | Negative 8 bit Integer      |
| n-int16    | xA9 | 2     | -      | Negative 16 bit Integer     |
| n-int32    | xAA | 4     | -      | Negative 32 bit Integer     |
| n-int64    | xAB | 8     | -      | Negative 64 bit Integer     |
| n-bignum16 | xAF | -     | 2      | Negative Bignum             |
| binary64   | x92 | 8     | -      | IEEE 754 Floating Point     |
|            |     |       |        | Binary 64 bit               |
| b-value    | xB0 | -     | -      | True                        |
| b-value    | xB1 | -     | -      | False                       |
| b-value    | xB2 | -     | -      | Null                        |
+------------+-----+-------+--------+-----------------------------+
```

Table 2

Table 2: Codes for Integers, 64 Bit Floating Point, Boolean and Null
items.

A data type commonly used in networking that is not defined in this
scheme is a datetime representation. To define such a data type, a
string containing a date-time value in Internet type format is
typically used.

4.1. JSON-B Examples

The following examples show examples of using JSON-B encoding:

A0 2A   42 (as 8 bit integer)

Figure 61

A1 00 2A   42 (as 16 bit integer)

Figure 62

A2 00 00 00 2A   42 (as 32 bit integer)

Figure 63

```
A3 00 00 00 00 00 00 00 2A        42 (as 64 bit integer)
```

Figure 64

```
A5 00 01 42                       42 (as Bignum)
```

Figure 65

Figure 66

```
80 05 48 65 6c 6c 6f              "Hello" (single chunk)
```

Figure 67

```
81 00 05 48 65 6c 6c 6f           "Hello" (single chunk)
```

Figure 68

```
84 05 48 65 6c 6c 6f 80 00        "Hello" (as two chunks)
```

Figure 69

Figure 70

```
92 3f f0 00 00 00 00 00 00        1.0
```

Figure 71

```
92 40 24 00 00 00 00 00 00        10.0
```

Figure 72

```
92 40 09 21 fb 54 44 2e ea        3.14159265359
```

Figure 73

```
92 bf f0 00 00 00 00 00 00          -1.0
```

Figure 74

Figure 75


```
B0                              true
```

Figure 76


```
B1                              false
```

Figure 77


```
B2                              null
```

Figure 78

5. JSON-C

JSON-C (Compressed) permits numeric code values to be substituted for
strings and binary data. Tag codes MAY be 8, 16 or 32 bits long
encoded in network byte order.

Tag codes MUST be defined before they are referenced. A Tag code MAY
be defined before the corresponding data or string value is used or
at the same time that it is used.

A dictionary is a list of tag code definitions. An encoding MAY
incorporate definitions from a dictionary using the dict-hash
production. The dict hash production specifies a (positive) offset
value to be added to the entries in the dictionary followed by the
UDF fingerprint [draft-hallambaker-udf] of the dictionary to be used.

```
+------------+-----+-------+--------+------------------------------+
| Production | Tag | Fixed | Length | Data Description             |
+------------+-----+-------+--------+------------------------------+
| c-tag      | xC0 | 1     | -      | 8 bit tag code               |
| c-tag      | xC1 | 2     | -      | 16 bit tag code              |
| c-tag      | xC2 | 4     | -      | 32 bit tag code              |
| c-def      | xC4 | 1     | -      | 8 bit tag definition         |
| c-def      | xC5 | 2     | -      | 16 bit tag definition        |
| c-def      | xC6 | 4     | -      | 32 bit tag definition        |
| c-tag      | xC8 | 1     | -      | 8 bit tag code and definition |
| c-tag      | xC9 | 2     | -      | 16 bit tag code and          |
|            |     |       |        | definition                   |
| c-tag      | xCA | 4     | -      | 32 bit tag code and          |
|            |     |       |        | definition                   |
| c-def      | xCC | 1     | -      | 8 bit tag dictionary         |
|            |     |       |        | definition                   |
| c-def      | xCD | 2     | -      | 16 bit tag dictionary        |
|            |     |       |        | definition                   |
| c-def      | xCE | 4     | -      | 32 bit tag dictionary        |
|            |     |       |        | definition                   |
| dict-hash  | xD0 | 4     | 1      | UDF fingerprint of dictionary |
+------------+-----+-------+--------+------------------------------+
```

                                Table 3

   Table 3: Codes Used for Compression

   All integer values are encoded in Network Byte Order (most
   significant byte first).

5.1. JSON-C Examples

   The following examples show examples of using JSON-C encoding:

   C8 20 80 05 48 65 6c 6c 6f   "Hello"   20 = "Hello"

                                Figure 79

   C4 21 80 05 48 65 6c 6c 6f   21 = "Hello"

                                Figure 80

   C0 20   "Hello"

                                Figure 81

   C1 00 20   "Hello"

                            Figure 82

                            Figure 83

   D0 00 00 01 00 20   Insert dictionary at code 256

                            Figure 84

   e3 b0 c4 42 98 fc 1c 14

                            Figure 85

   9a fb f4 c8 99 6f b9 24

                            Figure 86

   27 ae 41 e4 64 9b 93 4c

                            Figure 87

   a4 95 99 1b 78 52 b8 55   UDF (C4 21 80 05 48 65 6c 6c 6f)

                            Figure 88

6. JSON-D (Data)

   JSON-B and JSON-C only support the two numeric types defined in the
   JavaScript data model: Integers and 64 bit floating point values.
   JSON-D (Data) defines binary encodings for additional data types that
   are commonly used in scientific applications. These comprise
   positive and negative 128 bit integers, six additional floating point
   representations defined by IEEE 754 [IEEE754] and the Intel extended
   precision 80 bit floating point representation [INTEL] .

   Should the need arise, even bigger bignums could be defined with the
   length specified as a 32 bit value permitting bignums of up to 2^35
   bits to be represented.

    d-value = d-integer | d-float

                            Figure 89

                            Figure 90

    d-float = binary16 | binary32 | binary128 | binary80 |

                            Figure 91

    decimal32 | decimal64 | decimal 128

                            Figure 92

The codes for these values are as follows:

Production	Tag	Fixed	Length	Data Description
p-int128	xA4	16	-	Positive 128 bit Integer
n-int128	xAC	16	-	Negative 128 bit Integer
binary16	x90	2	-	IEEE 754 Floating Point Binary 16 bit
binary32	x91	4	-	IEEE 754 Floating Point Binary 32 bit
binary128	x94	16	-	IEEE 754 Floating Point Binary 64 bit
Intel80	x95	10	-	Intel extended Floating Point 80 bit
decimal32	x96	4	-	IEEE 754 Floating Point Decimal 32
Decimal64	x97	8	-	IEEE 754 Floating Point Decimal 64
Decimal128	x98	16	-	IEEE 754 Floating Point Decimal 128

                            Table 4

Table 4: Additional Codes for Scientific Data

[7](). **JBCD Frames and Records**

Tag codes in the range xF0-XFF are reserved for specifying markers
for frames and records. These tags are not used to encode JSON data,
they are only used to encapsulate opaque binary data blobs as a unit.

A JBCD record consists of consist of the tag, a length and the data
item. The length indication provided by the record format allows
efficient traversal of a sequence of records in the forward direction
only.

A JBCD Frames consists of consist of the tag, a length and the data
item followed by the tag-length sequence repeated with the bytes
written in the reverse order. The first length indication allows
efficient traversal of a sequence of records in the forward direction
and the second allows efficient traversal in the reverse direction.

[[This figure is not viewable in this format. The figure is
available at http://prismproof.org/Documents/draft-hallambaker-
jsonbcd.html [3].]]

JBCD Records and Frames

The JBCD-Frame tags currently defined are:

```
+------------+---------+-------+--------+----------------------+
| Production | Tag     | Fixed | Length | Data Description     |
+------------+---------+-------+--------+----------------------+
| uframe     | xF0     | -     | 1      | Record, 8 bit length |
| uframe     | xF1     | -     | 2      | Record, 16 bit length|
| uframe     | xF2     | -     | 4      | Record, 32 bit length|
| uframe     | xF3     | -     | 8      | Record, 64 bit length|
| bframe     | xF4     | -     | 1      | Frame, 8 bit length  |
| bframe     | xF5     | -     | 2      | Frame, 16 bit length |
| bframe     | xF6     | -     | 4      | Frame, 32 bit length |
| bframe     | xF7     | -     | 8      | Frame, 64 bit length |
|            | xF8-xFF | -     | -      | Reserved             |
+------------+---------+-------+--------+----------------------+
```

Table 5

The author does not expect additional framing tags to be added but
codes F8-FF are reserved in case this is desired.

It may prove convenient to represent message digest values as large
integers rather than binary strings. While very few platforms or
programming languages support mathematical operations on fixed size

integers larger than 64, this is not a major concern since message
digests are rarely used for any purpose other than comparison for
equality.

```
+------------+-----+-------+--------+-------------------------+
| Production | Tag | Fixed | Length | Data Description        |
+------------+-----+-------+--------+-------------------------+
| p-int128   | Xa4 | 16    | -      | Positive 128 bit Integer |
| p-int256   | Xa5 | 32    | -      | Positive 256 bit Integer |
| p-int512   | Xa6 | 64    | -      | Positive 512 bit Integer |
+------------+-----+-------+--------+-------------------------+
```

Table 6

8. Acknowledgements

This work was assisted by conversations with Nico Williams and other
participants on the applications area mailing list.

9. Security Considerations

A correctly implemented data encoding mechanism should not introduce
new security vulnerabilities. However, experience demonstrates that
some data encoding approaches are more prone to introduce
vulnerabilities when incorrectly implemented than others.

In particular, whenever variable length data formats are used, the
possibility of a buffer overrun vulnerability is introduced. While
best practice suggests that a coding language with native mechanisms
for bounds checking is the best protection against such errors, such
approaches are not always followed. While such vulnerabilities are
most commonly seen in the design of decoders, it is possible for the
same vulnerabilities to be exploited in encoders.

A common source of such errors is the case where nested length
encodings are used. For example, a decoder relies on an outermost
length encoding that specifies a length on 50 bytes to allocate
memory for the entire result and then attempts to copy a string with
a declared length of 1000 bytes within the sequence.

The extensions to the JSON encoding described in this document are
designed to avoid such errors. Length encodings are only used to
define the length of x-value constructions which are always terminal
and cannot have nested data entries.

10. IANA Considerations

   [TBS list out all the code points that require an IANA registration]

11. References

11.1. Normative References

   [draft-hallambaker-udf]
              Hallam-Baker, P., "Uniform Data Fingerprint (UDF)", draft-
              hallambaker-udf-08 (work in progress), October 2017.

   [IEEE754] IEEE Computer Society, "IEEE Standard for Floating-Point
              Arithmetic", IEEE 754-2008,
              DOI 10.1109/IEEESTD.2008.4610935, August 2008.

   [INTEL]   Intel Corp., "Unknown".

   [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data
              Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March
              2014.

11.2. Informative References

   [draft-hallambaker-mesh-architecture]
              Hallam-Baker, P., "Mathematical Mesh: Architecture",
              draft-hallambaker-mesh-architecture-04 (work in progress),
              September 2017.

11.3. URIs

   [1] http://prismproof.org/Documents/draft-hallambaker-jsonbcd.html

   [2] http://prismproof.org/Documents/draft-hallambaker-jsonbcd.html

   [3] http://prismproof.org/Documents/draft-hallambaker-jsonbcd.html

Author's Address

   Phillip Hallam-Baker
   Comodo Group Inc.

   Email: philliph@comodo.com