Javascript Object Notation (JSON) Encoding Rules for ASN.1

Objective Systems, Inc.

December 25, 2018

Author's Contact Information
Comments, suggestions, and inquiries regarding this document may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

Summary

This document is of interest only for historical purposes. It has been supplanted by ITU-T X.697 JSON Encoding Rules (JER). Differences between the encoding rules specified in this document and those specified in X.697 are noted in the section Comparison to X.697.

This document specifies rules for encoding values of ASN.1 types using JSON. These encoding rules have been specified by Objective Systems, Inc. This specification was influenced by ITU-T X.693 XML Encoding Rules (XER).

Normative References

The following references are considered normative for this specification.

- ITU-T Recommendation X.680 (2008) | ISO/IEC 8824-1:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.
- ITU-T Recommendation X.681 (2008) | ISO/IEC 8824-2:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.
- ITU-T Recommendation X.682 (2008) | ISO/IEC 8824-3:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2008) | ISO/IEC 8824-4:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.
- ECMA-262 (2011), *ECMAScript Language Specification*
- IETF RFC 4627, *The application/json Media Type for JavaScript Object Notation (JSON)*

General Rules

This section describes the general rules for encoding values of ASN.1 types in JSON.

This document uses productions from ITU-T X.680 as much as possible. Note in particular that `xmlhstring` and `xmlbstring` are used. There is nothing particularly "XML" about these productions and they conveniently describe the desired string.

`QUOTEMARK` is used wherever a quotation mark (") appears in the JSON text.

Encoding On the Wire

JSON values are strings of Unicode characters that conform to the JSON grammar. Thus, JSON encoding rules inherently specify a character-based encoding. However, for transmission, those characters will need to be represented in bytes. For that purpose, this specification follows IETF RFC 4627. According to RFC 4627, JSON shall be encoded in one of the following Unicode character encodings: UTF-8, UTF-16BE, UTF-16LE, UTF-32LE, UTF-32BE. The first four octets of an encoding can be examined to determine the specific character encoding (as explained in RFC 4627 section 3). Support for UTF-8 encoding is treated as mandatory and support for the other character encodings as optional.

JSONValue

The JSON encoding of an ASN.1 value shall be the `JSONValue` for that value. Every ASN.1 abstract value has an associated `JSONValue`. Every `JSONValue` must be a valid JSON value[1].

```
JSONValue ::=
      JSONBooleanValue |
      JSONIntegerValue |
      JSONEnumeratedValue |
      JSONRealValue |
      JSONBitStringValue |
      JSONOctetStringValue |
      JSONNullValue |
      JSONSequenceValue |
      JSONSequenceOfValue |
      JSONChoiceValue |
      JSONObjectIdentifierValue |
      JSONRelativeOIDValue |
      IRIValue |
```

1 This is important for the definition of `JSONNamedValue`.

```
            RelativeIRIValue |
            JSONEmbeddedPDVValue |
            JSONExternalValue |
            JSONTimeValue |
            JSONRestrictedCharacterStringValue |
            JSONUnrestrictedCharacterStringValue |
            JSONOpenTypeValue
```

`IRIValue` and `RelativeIRIValue` are defined in X.680. The remaining alternatives for `JSONValue` are defined in this document.

A `JSONValue` may contain insignificant whitespace as permitted by JSON, viz. around the delimiter characters: `{}[]:,`

JSONNestedValue

The specification for `JSONOpenTypeValue`, provides two alternatives for encoding an open type. The first alternative (`JSONNestedValue`) embeds the JSON text for the open type in a JSON object. The second alternative embeds a hexadecimal representation of the JSON text for the open type in a JSON string. These alternatives can be easily distinguished from each other, since one is a JSON object and the other is a JSON string. Within JavaScript, a programmer can use "instanceof" to determine whether the value is an Object or not (if not, then assume string).

```
      JSONNestedValue ::=
            "{" QUOTEMARK "value" QUOTEMARK ":" JSONValue "}"
```

`JSONNestedValue` serves much the same purpose as `XMLTypedValue` does in XER. `XMLTypedValue`, however, includes the ASN.1 type name, whereas `JSONNestedValue` does not. This is intentional. It is not always very clear what name should appear in an `XMLTypedValue` and in some cases the name is not useful for decoding anyway. `JSONNestedValue` does not include the ASN.1 type name to avoid confusion and to prevent someone from incorrectly relying on it to determine the type of the encoded value.

Mapping Identifiers

Names used in the JSON should be valid JavaScript identifier names[2]. Using something other than a valid JavaScript identifier name for a property name forces programmers to use object["name"] notation rather than object.name notation and may also prevent some optimizations in the JavaScript engine.

ASN.1 identifiers are mapped to JSON identifier names when encoding sequence, set, and choice

2 Identifiers cannot be reserved words, but identifier names can be. For example, "break" is a reserved word but is a valid property name. So you can use either myobject.break or myobject["break"].

types. Not every ASN.1 identifier is a valid JSON identifier name. An ASN.1 identifier is also a valid JavaScript identifier name if and only if it does not contain any hyphens. This makes the conversion rule simple:

- convert any hyphen into an underscore

This maps every possible ASN.1 identifier to a distinct JavaScript identifier name. Conveniently, ASN.1 identifier names may not contain underscores. Thus every underscore in the transformed identifier comes from applying the above rule.

The result of the above mapping produces a `jsonasn1identifier` from an ASN.1 identifier.

Encoding by ASN.1 Type

BOOLEAN

```
JSONBooleanValue ::= "true" | "false"
```

INTEGER

```
JSONIntegerValue ::= SignedNumber
```

`INTEGER` types with named numbers receive no special treatment. This seemed preferable on the assumption that the JSON text would be operated on by JavaScript directly.

ENUMERATED

```
JSONEnumeratedValue ::= number
```

The encoded number shall be the value associated with the enumerated value.

REAL

```
JSONRealValue ::=
            NumericRealValue
      |     QUOTEMARK JSONSpecialRealValue QUOTEMARK

JSONSpecialRealValue ::=
      "NaN" | "NEGATIVE_INFINITY" | "POSITIVE_INFINITY"
```

The alternatives for `JSONSpecialRealValue` correspond to the names used in JavaScript for these values. There does not seem to be any method in JavaScript that takes these strings and returns a corresponding Number value (e.g. Number.NEGATIVE_INFINITY). In fact, the parseFloat method will return Number.NaN for any string that does not begin with a number. However, representing these special values as strings is the best that can be done under JSON.

BIT STRING

```
JSONBitStringValue ::= QUOTEMARK xmlbstring QUOTEMARK
```

`BIT STRING` types having named bits receive no special handling. An enhancement would be to allow a value with only named bits set to have an alternative representation that used the names of the named bits in the JSON value. It is not clear whether a JSON object or JSON array would be more convenient to a Javascript programmer. Such a feature did not seem especially worthwhile.

`BIT STRING` types with contents constraints receive no special handling. An enhancement would be to allow a `JSONNestedValue` representation.

Note that a contents constraint without `ENCODED BY` would imply that the value of the bit string is the JSON encoding of some abstract value of the specified type. That means it should be the Unicode encoding (UTF-8, UTF-16LE, UTF-16BE, etc.) of a `JSONValue`. The particular Unicode encoding used shall be the same as that used for the bit string itself.

OCTET STRING

```
JSONOctetStringValue ::= QUOTEMARK xmlhstring QUOTEMARK
```

`OCTET STRING` types with contents constraints receive no special handling. An enhancement would be to allow a `JSONNestedValue` representation.

Note that a contents constraint without `ENCODED BY` would imply that the value of the octet string is the JSON encoding of some abstract value of the specified type. That means it should be the Unicode encoding (UTF-8, UTF-16LE, UTF-16BE, etc.) of a `JSONValue`. The particular Unicode encoding used shall be the same as that used for the octet string itself.

NULL

```
JSONNullValue ::= "null"
```

SEQUENCE and SET

A sequence value is modeled as a JSON Object[3]. There is no difference in the encoding for a set and sets will not be referred to further.

```
JSONSequenceValue ::=
                "{"  "}"
        |       {" JSONComponentValueList "}"
```

3 The choice of JSON object instead of JSON array was to make the encoding more friendly to Javascript programmers and to human readers.

```
JSONComponentValueList ::=
            JSONNamedValue
      |     JSONComponentValueList "," JSONNamedValue

JSONNamedValue ::= QUOTEMARK jsonasn1identifier QUOTEMARK ":" JSONValue
```

There shall be a `JSONNamedValue` in the `JSONSequenceValue` for every `NamedType` in the sequence which is not marked `OPTIONAL` or `DEFAULT`.

Decoders must be prepared to handle an unknown `JSONNamedValue` resulting from extensions.

Note especially that a JSON object's name-value pairs are unordered. Since X.680 25.14 requires the identifiers to be distinct, this is not a problem. Decoders must be prepared to accept the name-value pairs in any order.

The `jsonasn1identifier` will be determined as described in the section on mapping identifiers.

SEQUENCE OF and SET OF

A sequence-of value is modeled as a JSON array. There is no difference in the encoding for a set-of and set-of's will not be referred to further.

```
JSONSequenceOfValue ::=
            "["  "]"
      |     "[" JSONValueList "]"

JSONValueList ::=
            JSONValue
      |     JSONValueList "," JSONValue
```

CHOICE

A choice value is modeled as a JSON Object having exactly one or zero properties (a choice type with no alternatives produces zero properties).

```
JSONChoiceValue ::=
            "{"  "}"
      |     "{" JSONNamedValue "}"
```

For `JSONNamedValue`, see the section on `SEQUENCE`.

Within JavaScript, a programmer can simply check each of the possible properties and find which of them is not undefined.

Object Identifier

```
JSONObjectIdentifierValue ::= QUOTEMARK XMLObjectIdentifierValue QUOTEMARK
```

Relative Object Identifier

```
JSONRelativeOIDValue ::= QUOTEMARK XMLRelativeOIDValue QUOTEMARK
```

Embedded-pdv

```
JSONEmbeddedPDVValue ::= JSONSequenceValue
```

The encoded value for an embedded PDV shall be the encoding of the sequence given in X.680 36

External

```
JSONExternalValue ::= JSONSequenceValue
```

The encoded value for an external value shall be the encoding of the sequence given in X.680 37

Time

```
JSONTimeValue ::= tstring
```

N.B. tstring is defined in X.680 and includes opening and closing quotes

Restricted Character String

```
JSONRestrictedCharacterStringValue ::= QUOTEMARK characters QUOTEMARK
```

"characters" will be each of the characters from the string. Any characters which cannot be represented directly in a JSON string value will be represented using one of the allowable escape sequences for a JSON string.

Unrestricted Character String

```
JSONUnrestrictedCharacterStringValue ::= JSONSequenceValue
```

The encoding for an unrestricted character string shall be the encoding of the sequence given in X.680

44.5.

Generalized time, Universal time, Object Descriptor

These ASN.1 types are all defined in terms of ASN.1 restricted character string types. They shall be encoded as such.

Open Type

There are two alternatives for encoding an open type, similar to the handling of open type in XML, as specified for `XMLOpenTypeFieldVal` (see X.681).

```
    JSONOpenTypeValue ::=
                JSONNestedValue
          |     QUOTEMARK xmlhstring QUOTEMARK
```

If the `JSONNestedValue` alternative is used, then its `JSONValue` shall be the `JSONValue` corresponding to the actual type.

If the second alternative of `JSONOpenTypeValue` is used, then the `xmlhstring` shall be the hexadecimal representation of an encoding of the abstract value according to some unspecified encoding rules. This alternative is not generally recommended but can be useful when decoding from some other encoding rules and encoding to JSON.

Example

```
JERExample
DEFINITIONS
AUTOMATIC TAGS
::= BEGIN

Records ::= SEQUENCE OF record Record

Record ::= SEQUENCE {
   mainInfo SEQUENCE {
      a-boolean BOOLEAN,
      an-integer INTEGER,
      status ENUMERATED {new, active, closed},
      a-real REAL,
      bitstr BIT STRING,
      flags-in-bitstr BIT STRING (CONTAINING (Flags)),
      octstr OCTET STRING,
      flags-in-octstr OCTET STRING (CONTAINING (Flags)),
      nothing NULL,
      whichone ThisOrThat,
      message UTF8String
    } OPTIONAL,
    real2 REAL OPTIONAL,
    int2 INTEGER DEFAULT 10,
    an-open-type TYPE-IDENTIFIER.&Type OPTIONAL
}

Flags ::= SEQUENCE {
   flag1 BOOLEAN,
   flag2 BOOLEAN
}
ThisOrThat ::= CHOICE {
   this INTEGER,
   that UTF8String
}
```

```
--The value that will be shown in JSON.
theValue Records ::= {
   record {
      mainInfo {
         a-boolean TRUE,
         an-integer 9,
         status active,
         a-real 34.58,
         bitstr '01110110111'B,
         flags-in-bitstr CONTAINING {flag1 TRUE, flag2 FALSE},
         octstr '2BE02BE'H,
         flags-in-octstr CONTAINING { flag1 TRUE, flag2 FALSE},
         nothing NULL,
         whichone this : 25,
         message "smell the roses"
      },
      int2 29,
      an-open-type INTEGER(0..20) : 16
   },
   record {
      real2 PLUS-INFINITY,
      an-open-type INTEGER(0..20) : 16
   }
}

END
```

The JSON encoding would be a Unicode encoding (UTF-8, say) of the following text:

```
[
   { "mainInfo" :
      {
         "a_boolean" : true,
         "an_integer" : 9,
         "status" :  1,
         "a_real" : 34.58,
         "bitstr" : "01110110111",
         "flags_in_bitstr" :
"0111101100100010011001100110110001100001011001110011000100100010001110100111010001
11001001110101011001010010110000100010011001100110110001100001011001110011001000100
01000111010011001100110000101101100011100110110010101111101",
         "octstr" : "2BE02BE",
         "flags_in_octstr" :
"7B22666C616731223A747275652C22666C616732223A66616C73657D",
         "nothing" : null,
         "whichone" : { "this" : 25 },
         "message" : "smell the roses"
      },
      "int2" : 29,
      "an_open_type" : "3136"
   },
   {
      "real2" : "POSITIVE_INFINITY",
      "an_open_type" : { "value" : 16 }
   }
]
```

NOTE 1: The character sequence:

```
  {"flag1":true,"flag2":false}
```

which is the `JSONValue` for a `Flags` value, encoded in UTF-8 is:

```
hexadecimal: 7B22666C616731223A747275652C22666C616732223A66616C73657D
binary:
01111011001000100110011001101100011000010110011100110001001000100011101001110100011
10010011101010110010100101100001000100110011001101100011000010110011100110010001000
1000111010011001100110000101101100011100110110010101111101
```

These strings appear in the "flags_in_bistr" and flags_in_octstr" fields.

NOTE 2: The character sequence:

```
   16
```

which is the `JSONValue` for an `INTEGER` value of 16, encoded in UTF-8 is:

```
hexadecimal: 3136
```

In the above example, you can see a few things demonstrated, in addition to the encoding of various ASN.1 types:

- identifier mapping ("-" becomes "_")
- encoding of open type using both alternatives. A single message would most likely use only one alternative, but both are shown here for comparison.

Comparison to X.697

This section compares the encoding rules specified in this document with those specified in ITU-T X.697-201710. This section might not mention every difference and does not comprehensively describe the differences.

- X.697 uses ASN.1 identifiers without modification in the encoding. It does not convert hyphens to underscores.
- X.697 uses the identifier for ENUMERATED rather than the associated (numeric) enumerated value.
- X.697 encodes REAL values differently. It allows for a distinction between base2 and base10 REAL values (which are two distinct sets of abstract values), makes use of constraints in determining the encoding, encodes -0 as a string, and uses different strings for the inifinities.
- X.697 encodes BIT STRING differently. The encoding varies based on what constraints are applied. It does not use xmlbstring at all.
- X.697 encodes certain restricted character string types differently, as if the strings were octet strings. These include: TeletexString, T61String, VideotexString, GraphicString, and GeneralString.
- X.697 encodes an open types as if encoding the actual type for the open type. It does not wrap the open type nor does it allow for a hexadecimal string in order to carry non-JER-encoded data.
- X.697 allows uses to alter the encoding by using encoding control notation. Our proprietary rules did not provide this feature. This is not an issue when moving from our propriety rules to X.697.