Overview of the JSON Encoding Rules (JER)

Alessandro Triglia, OSS Nokalva

sandro@oss.com

July 2017

CONTENTS

1 Introduction

This paper is a technical overview of the JSON Encoding Rules (JER), a set of encoding rules for ASN.1 recently developed by OSS Nokalva and submitted to the ISO/IEC and ITU-T for standardization. The JER specification is expected to become an ITU-T Recommendation and be made freely available for download in early 2018.

JER was designed to allow applications to represent any ASN.1 value in JavaScript Object Notation (JSON). For example, an application that receives BER- or PER-encoded messages conforming to a certain ASN.1 schema will be able to re-encode the messages in JSON. This capability can be used to exchange data with other applications that do not understand ASN.1 but know how to parse or write JSON.

JER is analogous to the XML Encoding Rules of ASN.1 (XER) in that whereas XER produces an XML document that represents a given ASN.1 value and can be read by any XML parser, JER produces a JSON text that represents a given ASN.1 value and can be read by any JSON parser. In both cases, the encoding produced is textual and human-readable, in contrast with the traditional encoding rules of ASN.1 (e.g., BER, DER, PER, OER) which produce binary encodings.

The JER specification also defines a small set of encoding instructions (JER encoding instructions), which can be included in an ASN.1 schema to modify in specific ways the JSON text representing ASN.1 values.

JER allows ASN.1 to be used as a schema language for JSON. A variety of commonly occurring JSON structures can be specified using ASN.1 with or without JER encoding instructions.

This paper describes the JER encodings of a few major ASN.1 types. It also describes JER encoding instructions and their effect on the JER encodings.

2 The JSON Encoding Rules

The JSON Encoding Rules (JER) encode any value of an ASN.1 type as JSON text. They can be applied to any ASN.1 schema and are designed to produce JSON text that will look natural to a typical JSON user.

For most ASN.1 types, the JER encodings are straightforward. For example, in the absence of JER encoding instructions (see below), `SEQUENCE` types are encoded as JSON objects, `SEQUENCE OF` types are encoded as JSON arrays, `INTEGER`s are encoded as JSON numbers, and `BOOLEAN`s are encoded as JSON `true` and `false`.

In JER, each ASN.1 type has a default JSON representation. The schema author can modify in specific ways the JSON representation of some types defined in an ASN.1 schema by using JER encoding instructions (see below).

In JER there are usually several ways to encode a given value. This is due to the following facts:

- there are no restrictions on the kind of white space characters occurring between JSON tokens
- the properties of a JSON object may appear in any order
- escapes of the form \u0000 through \uFFFF may be freely used in any JSON string.

2.1 JER encoding instructions

JER encoding instructions are a special notation that can be included in an ASN.1 schema and modify the JSON text that JER produces when it's applied to that schema. Not every ASN.1/JER schema will need to contain JER encoding instructions; if the JSON text produced by JER by default is acceptable for the purposes of the schema, no encoding instructions are necessary.

The presence of JER encoding instructions in an ASN.1 module does not prevent the application of other standard encoding rules such as PER or BER to that module, nor does it affect the encodings produced by other encoding rules.

The JER specification indicates which JER encoding instructions are applicable to each ASN.1 type and describes their effect on the JER encodings.

A JER encoding instruction is enclosed in square brackets (e.g., `[ARRAY]`) and can be placed either in front of an ASN.1 type (like a tag) or in a special section of the ASN.1 module called the "JER encoding control section" (see examples below). A JER encoding instruction that is prefixed to a type applies only to that type. A JER encoding instruction present in the JER encoding control section of a module applies to multiple types defined in that module. For example, an encoding instruction `[BASE64 OCTET STRING]` included in the JER encoding control section causes all the octet string types present in the same module to be encoded as a base-64 string instead of a hexadecimal string (see below).

The following sections describe the JER encodings of a few major ASN.1 types and the JER encoding instructions that can be applied to them.

2.2 Encoding of a boolean value

A boolean value is encoded as one of the JSON tokens `false` and `true` (without quotes), denoting the values `FALSE` and `TRUE`, respectively.

2.3 Encoding of an integer value

An integer value is encoded as a JSON number (without quotes) with no fractional part and no exponent.

2.4 Encoding of an enumerated value

The encoding of enumerated values depends on the presence of a `[TEXT]` encoding instruction applied to the enumerated type.

In the absence of `[TEXT]` encoding instructions, a value of an enumerated type is encoded as a JSON string (with quotes) which is the name of the enumeration item surrounded by quotes.

A `[TEXT]` encoding instruction applied to an enumerated type changes the encoding of one or more of its enumeration items. The schema author can change the initial letter of an item's name from lowercase to uppercase by using `[TEXT item-name AS CAPITALIZED]`, change the case of all the letters of an item's name by using `[TEXT item-name AS UPPERCASED` or `LOWERCASED]`, or replace an item's name with a different string by using `[TEXT item-name AS "new string"]`. The notation `[TEXT ALL AS CAPITALIZED`, etc.`]` can be used to target all the enumeration items.

2.5 Encoding of a real value

In ASN.1, the values of the `REAL` type can be divided into three subsets—base-2 real values, base-10 real values, and special real values.

Base-2 real values are encoded as JSON numbers.

The special real values are encoded as JSON strings, as follows:

`0`	`"0"`
`-0`	`"-0"`
`MINUS-INFINITY`	`"-INF"`
`PLUS-INFINITY`	`"INF"`
`NOT-A-NUMBER`	`"NaN"`

Base-10 real values are encoded in one of the following two forms, depending on the constraints present in the definition of the real type. If the real type is unconstrained or the set of its permitted values includes both base-2 and base-10 real values, then a base-10 real value is encoded as a JSON object with a single property `"base10Value"` whose value is a JSON number. If the constraints present in the definition of the real type forbid all base-2 values, then a base-10 real value is encoded as a JSON number.

In most practical cases, only the simpler form of real value encoding (a JSON number) will occur.

2.6 Encoding of a string value

A value of most ASN.1 character string types (`IA5String`, `VisibleString`, `PrintableString`, `BMPString`, `UniversalString`, `UTF8String`, etc.) is encoded as a JSON string (with quotes).

Note that escapes of the form \u0000 through \uFFFF are allowed anywhere within a JSON string.

2.7 Encoding of an octet string value

The encoding of octet string values depends on the presence of a [BASE64] encoding instruction applied to the octet string type.

In the absence of a [BASE64] encoding instruction, a value of an octet string type is encoded as a JSON string (with quotes) consisting of an even number of the hexadecimal digits 0123456789abcdefABCDEF. Each pair of hexadecimal digits encodes one octet of the octet string value.

When a [BASE64] encoding instruction is applied to an octet string type, each value of that type is encoded as a base-64 string.

2.8 Encoding of a bit string value

A bit string value is encoded in one of the two forms shown below, depending on the constraints present in the definition of the type.

The first form, used when the bit string type has a fixed size, is a JSON string (with quotes) consisting of an even number of the hexadecimal digits 0123456789abcdefABCDEF. Each pair of hexadecimal digits encodes eight bits of the bit string value. If the size of the bit string is not a multiple of 8, it is encoded as if it contained extra bits up to the next multiple of 8, all set to zero.

The second form, used when the bit string type has a variable size, is a JSON object with the following structure:

```
{
        "value": <a JSON string constructed as described above>,
        "length": <the length of the bit string as a JSON number (without quotes)>
}
```

2.9 Encoding of a sequence value

The encoding of sequence values depends on the presence of an [ARRAY] encoding instruction applied to the sequence type.

In the absence of an [ARRAY] encoding instruction, a value of a sequence type is encoded as a JSON object having one property for each component of the sequence type that is present in the value. The name of the property is, by default, the identifier of the component, and the value of the property is the JER encoding of the value of the component.

The name of the JSON object property for a component of the sequence type can be changed by applying a [NAME] encoding instruction to the component. The schema author can change the initial letter of the component's name from lowercase to uppercase by using [NAME AS CAPITALIZED], change the case of all the letters of the component's name by using [NAME AS UPPERCASED or LOWERCASED], or replace the component's name with a different string by using [NAME AS "*new string*"].

If the sequence type has some OPTIONAL or DEFAULT components, for each such component that is absent in the sequence value (with some exceptions described in the JER specification), the encoder may include in the

JSON object an additional property whose value is set to `null` (this has the same meaning as not including the property at all).

Note that the properties of a JSON object may occur in any order.

When an `[ARRAY]` encoding instruction is applied to a sequence type, each value of that type is encoded as a JSON array that has one element for each component of the sequence type. Any `OPTIONAL` or `DEFAULT` components of the sequence type that are absent in the value are encoded as JSON `null`. The encoder may omit any trailing array elements that are set to `null`.

2.10 Encoding of a set value

Set values are encoded like sequence values, except that the `[ARRAY]` encoding instruction is not allowed on a set type.

2.11 Encoding of a sequence-of value

The encoding of a sequence-of value is a JSON array having one element for each occurrence of the repeating component. The value of each element of the array is the JER encoding of the value of the corresponding occurrence.

2.12 Encoding of a set-of value

The encoding of set-of values depends on the presence of an `[OBJECT]` encoding instruction applied to the set-of type.

In the absence of an `[OBJECT]` encoding instruction, a value of a set-of type is encoded as a JSON array having one element for each occurrence of the repeating component. The value of each element of the array is the JER encoding of the value of a different occurrence. The elements may be added in any order.

When an `[OBJECT]` encoding instruction is applied to a set-of type, each value of that type is encoded as a JSON object. This encoding instruction may be applied only when the component of the set-of type is a sequence type with two components, which are treated as a "key" and a "value". As a value of a set-of type contains, in general, multiple occurrences of the component, there will be multiple such "key-value pairs" and each of them will appear as a property of the resulting JSON object.

2.13 Encoding of a choice value

The encoding of choice values depends on the presence of an `[UNWRAPPED]` encoding instruction applied to the choice type.

In the absence of an `[UNWRAPPED]` encoding instruction, a value of a choice type is encoded as a JSON object having exactly one property. The name of the property is, by default, the identifier of the chosen alternative, and the value of the property is the JER encoding of the value of the alternative.

The name of the property for an alternative of the choice type can be changed by applying a `[NAME]` encoding instruction to the alternative. The schema author can change the initial letter of the alternative's name from lowercase to uppercase by using `[NAME AS CAPITALIZED]`, change the case of all the letters of the

alternative's name by using `[NAME AS UPPERCASED` or `LOWERCASED]`, or replace the alternative's name with a different string by using `[NAME AS "new string"]`.

When an `[UNWRAPPED]` encoding instruction is applied to a choice type, the encoding of a value of that type is identical to the encoding of the selected alternative, without a wrapping JSON object. The unwrapped encoding of choice types relies on the decoder's ability to identify the alternative that was encoded by examining the JER encoding of the alternative since there is no explicit indication of which alternative was encoded.

3 Example

3.1 A module with no JER encoding instructions

Suppose we have the following ASN.1 module:

```
MyModule-1 DEFINITIONS AUTOMATIC TAGS ::= BEGIN

A ::= SEQUENCE {
      a1    INTEGER (0..100),
      a2    INTEGER (-290..399),
      a3    INTEGER (0..60000)      OPTIONAL,
      a4    INTEGER (-5000000..5000000),
      a5    INTEGER (1000..MAX),
      a6    INTEGER (-1..MAX),
      a7    INTEGER                 OPTIONAL
}

a A ::= {
      a1 4,
      a2 4,
      a3 4,   --optional and present
      a4 4,
      a5 1024,
      a6 4
              --a7 is optional and absent
}

B ::= SEQUENCE {
      b1    UTF8String,
      b2    IA5String (SIZE (3)),
      b3    IA5String,
      b4    OCTET STRING,
      b5    BIT STRING (SIZE (4)),
      b6    BIT STRING
}

b B ::= {
      b1 "ABC",
      b2 "ABC",
      b3 "ABC",
      b4 '01020304'H,
      b5 '0101'B,
      b6 '1100'B
}

C ::= CHOICE {
      c1    BOOLEAN,
      c2    SEQUENCE OF ENUMERATED { a, b, c, d, e }
```

```
}

c C ::= c2 : { b, c, d, e }

END
```

Below we show some of the ways in which the value a can be encoded.

1 – here is a possible JER encoding of value a:

```
{
      "a1": 4,
      "a2": 4,
      "a3": 4,
      "a4": 4,
      "a5": 1024,
      "a6": 4
}
```

2 – here the missing optional component (a7) is explicitly encoded as null:

```
{
      "a1": 4,
      "a2": 4,
      "a3": 4,
      "a4": 4,
      "a5": 1024,
      "a6": 4,
      "a7": null
}
```

3 – here the properties of the resulting JSON object occur in a different order:

```
{
      "a5": 1024,
      "a6": 4,
      "a1": 4,
      "a4": 4,
      "a3": 4,
      "a2": 4
}
```

4 – here some of the JSON strings contain escapes:

```
{
      "a\u0035": 1024,
      "a6": 4,
      "\u00611": 4,
      "a4": 4,
      "a3": 4,
```

```
      "a2": 4
}
```

Below we show some of the ways in which the value b can be encoded.

1 – here is a possible JER encoding of value b:

```
{
      "b1": "ABC",
      "b2": "ABC",
      "b3": "ABC",
      "b4": "01020304",
      "b5": "50",
      "b6": { "length": 4, "value": "C0" }
}
```

2 – here the properties of both JSON objects occur in a different order:

```
{
      "b4": "01020304",
      "b2": "ABC",
      "b6": { "value": "C0", "length": 4 },
      "b3": "ABC",
      "b5": "50",
      "b1": "ABC"
}
```

3 – here some of the JSON strings contain escapes:

```
{
      "b4": "01020\u003304",
      "\u00622": "ABC",
      "b6": { "value": "\u0043\u0030", "length": 4 },
      "b3": "\u0041BC",
      "b5": "5\u0030",
      "b1": "ABC"
}
```

Below we show some of the ways in which the value c can be encoded.

1 – here is a possible JER encoding of value c:

```
{
      "c2": [ "b", "c", "d", "e" ]
}
```

2 – here some of the JSON strings contain escapes:

```
{
      "c\u0032":
```

```
      [ "b",
        "\u0063",
        "d",
        "e" ]
}
```

3.2 A module with some JER encoding instructions

Suppose we have the following ASN.1 module (the JER encoding instructions are highlighted):

```
MyModule-2 DEFINITIONS JER INSTRUCTIONS AUTOMATIC TAGS ::= BEGIN

A ::= SEQUENCE {
      a1    INTEGER (0..100),
      a2    [NAME AS "_1/ (2@3&"] INTEGER (-290..399),
      a3    INTEGER (0..60000)      OPTIONAL,
      a4    OCTET STRING,
      a5    INTEGER                 OPTIONAL
}

a A ::= {
      a1 1,
      a2 2,
      a3 3,   --optional and present
      a4 '0102030405FFEE88AACC'H,
             --a5 is optional and absent
}

A2 = [ARRAY] A

a2 A2 ::= a

B ::= [OBJECT] SET OF SEQUENCE {
      k     IA5String,
      v     INTEGER (1..10000)
}

b B ::= {
      { k "one", v 551 },
      { k "two", v 1615 }
}

C ::= [UNWRAPPED] CHOICE {
      c1    BOOLEAN,
      c2    SEQUENCE OF [TEXT ALL AS CAPITALIZED] ENUMERATED { a, b, c, d, e }
}

c C ::= c2 : { b, c, d, e }

ENCODING-CONTROL JER
```

```
      [BASE64] OCTET STRING  --applies to all OCTET STRING types in the module
END
```

Here is a possible JER encoding of value a:

```
{
      "a1": 1,
      "_1/ (2@3&": 2,
      "a3": 3,
      "a4": "AQIDBAX/7oiqzA=="
}
```

Here is a possible encoding of value a2:

```
      [ 1, 2, 3, "AQIDBAX/7oiqzA==", null ]
```

Here is a possible encoding of value b:

```
{
      "one": 551,
      "two": 1615
}
```

Here is a possible encoding of value c:

```
       [ "B", "C", "D", "E" ]
```

About the author

Alessandro has been involved in the standardization of ASN.1 since 2001, and contributed most of the content of ITU-T X.694 (mapping from XML Schema to ASN.1), ITU-T X.891 (Fast Infoset), and ITU-T X.696 (Octet Encoding Rules). He has spoken about ASN.1 at several venues and has given tutorials on ASN.1 to various standards committees.