Encodings for oBIX: Common Encodings Version 1.0

Committee Specification Draft 01 / Public Review Draft 01

11 July 2013

Specification URIs

This version:

http://docs.oasis-open.org/obix/obix-encodings/v1.0/csprd01/obix-encodings-v1.0-csprd01.pdf (Authoritative)
http://docs.oasis-open.org/obix/obix-encodings/v1.0/csprd01/obix-encodings-v1.0-csprd01.html
http://docs.oasis-open.org/obix/obix-encodings/v1.0/csprd01/obix-encodings-v1.0-csprd01.doc

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/obix/obix-encodings/v1.0/obix-encodings-v1.0.pdf (Authoritative)
http://docs.oasis-open.org/obix/obix-encodings/v1.0/obix-encodings-v1.0.html
http://docs.oasis-open.org/obix/obix-encodings/v1.0/obix-encodings-v1.0.doc

Technical Committee:

OASIS Open Building Information Exchange (oBIX) TC

Chair:

Toby Considine (toby.considine@unc.edu), University of North Carolina at Chapel Hill

Editor:

Markus Jung (mjung@auto.tuwien.ac.at), Institute of Computer Aided Automation, Vienna University of Technology

Related work:

This specification is related to:

- *oBIX Version 1.1*. 11 July 2013. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/obix/obix/v1.1/csprd01/obix-v1.1-csprd01.html.
- *Bindings for oBIX: REST Bindings Version 1.0*. 11 July 2013. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/obix/obix-rest/v1.0/csprd01/obix-rest-v1.0-csprd01.html.
- *Bindings for oBIX: SOAP Bindings Version 1.0.* 11 July 2013. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/obix/obix-soap/v1.0/csprd01/obix-soap-v1.0-csprd01.html.

Abstract:

This document specifies different encodings for oBIX objects adhering to the oBIX object model. oBIX provides the core information model and interaction pattern for communication with building control systems. Specific implementations of oBIX must choose how to encode oBIX Information. The core specification describes an XML encoding, which is used in all examples in that document. This document specifies common alternate encodings, including CoAP, EXI, and JSON.

Status:

This document was last revised or approved by the OASIS Open Building Information Exchange (oBIX) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/obix/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/obix/ipr.php).

Citation format:

When referencing this specification the following citation format should be used:

[oBIX-Encodings]

Encodings for oBIX: Common Encodings Version 1.0. 11 July 2013. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/obix/obix-encodings/v1.0/csprd01/obix-encodings-v1.0-csprd01.html.

Notices

Table of Contents

1 Introduction

This document specifies the encodings for oBIX.

1.1 Terminology

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** Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt.

oBIX.

1.2 Normative References

RFC2119	Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt.
oBIX	*oBIX Version 1.1*. See link in “Related work” section on cover page.
EXI	J. Schneider, T. Kamiya, Efficient XML Interchange (EXI) Format 1.0, W3C Recommendation, 10 March 2011
RFC4627	Crockford, D., „The application/json Media type for JavaScript Object Notation (JSON), RFC, 4627, July 2007

1.3 Non-Normative References

REST	**RT Fielding** *Architectural Styles and the Design of Network-based Software Architectures,* Dissertation, University of California at Irvine, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
EXI MR	Y. Doi, EXI Messaging Requirements, IETF Internet-Draft, 25 February 2013
EXI BP	M. Cokus, D. Vogelheim, Efficient XML Interchange (EXI) Best Practices, W3C Working Draft, 19 December 2007

2 XML Encoding

This chapter specifies how the oBIX object model is encoded in XML.

2.1 Design Philosophy

Since there are many different approaches to developing an XML syntax, it is worthwhile to provide a bit of background to how the oBIX XML syntax was designed. Historically in M2M systems, non-standard extensions have been second class citizens at best, but usually opaque. One of the design principles of oBIX is to embrace vertical domain and vendor specific extensions, so that all data and services have a level playing field.

In order to achieve this goal, the XML syntax is designed to support a small, fixed schema for all oBIX documents. If a client agent understands this very simple syntax, then the client is guaranteed access to the server's object tree regardless of whether those objects implement standard or non-standard contracts.

Higher level semantics are captured via contracts. Contracts "tag" an object with a type and can be applied dynamically. This is very useful for modeling systems which are dynamically configured in the field. What is important is that contracts are optionally understood by clients. Contracts do not affect the XML syntax nor are clients required to use them for basic access to the object tree. Contracts are merely an abstraction which is layered cleanly above the object tree and its fixed XML syntax.

2.2 XML Syntax

The oBIX XML syntax maps very closely to the abstract object model. The syntax is summarized:

- Every oBIX object maps to exactly one XML element;
- An object's children are mapped as children XML elements;
- The XML element name maps to the built-in object type;
- Everything else about an object is represented as XML attributes;

The object model figure in Chapter 4 of the oBIX core specification [oBIX] illustrates the valid XML elements and their respective attributes. Note the `val` object is simply an abstract base type for the objects which support the `val` attribute - there is no `val` element.

2.3 XML Encoding

The following rules apply to encoding oBIX documents:

- oBIX documents MUST be well formed XML;
- oBIX documents SHOULD begin with XML Declaration specifying their encoding;
- It is RECOMMENDED to use UTF-8 encoding without a byte order mark;
- oBIX documents MUST NOT include a Document Type Declaration – oBIX documents cannot contain an internal or external subset;
- oBIX documents SHOULD include an XML Namespace definition. Convention is to declare the default namespace of the document to "http://obix.org/ns/schema/1.1". If oBIX is embedded inside another type of XML document, then the convention is to use "o" as the namespace prefix. Note that the prefix "obix" SHOULD NOT be used (see Section 2.6).

2.4 XML Decoding

The following rules apply to decoding of oBIX documents:

- MUST conform to XML processing rules as defined by XML 1.1;
- Documents which are not well formed XML MUST be rejected;
- Parsers are not required to understand a Document Type Declaration;
- Any unknown element MUST be ignored regardless of its XML namespace
- Any unknown attribute MUST be ignored regardless of its XML namespace

The basic rule of thumb is: strict in what you generate, and liberal in what you accept. oBIX parsers are required to ignore elements and attributes which they do not understand. However an oBIX parser MUST never accept an XML document which isn't well formed (such as mismatched tags).

2.5 XML Namespace

XML namespaces for standards within the oBIX umbrella should conform to the following pattern:

```
http://obix.org/ns/{spec}/{version}
```

The XML namespace for oBIX version 1.1 is:

```
http://obix.org/ns/schema/1.1
```

All XML in this document is assumed to have this namespace unless otherwise explicitly stated.

2.6 Namespace Prefixes in Contract Lists

XML namespace prefixes defined within an oBIX document may be used to prefix the URIs of a contract list. If a URI within a contract list starts with string matching a defined XML prefix followed by the ":" colon character, then the URI is normalized by replacing the prefix with its namespace value. This rule also applies to the `href` attribute as a convenience for defining contract themselves.

The XML namespace prefix of "obix" is predefined. This prefix is used for all the oBIX defined contracts. The "obix" prefix is literally translated into "http://obix.org/def/". For example the URI "obix:bool" is translated to "http://obix.org/def/bool". Documents SHOULD NOT define an XML namespace using the prefix "obix" which collides with the predefined "obix" prefix – if it is defined, then it is superseded by the predefined value of "http://obix.org/def/". All oBIX defined contracts are accessible via their HTTP URI using the HTTP binding (at least they should be one day).

An example oBIX document with XML namespace prefixes normalized:

```
<obj xmlns:acme="http://acme.com/def/" href="acme:CustomPoint"
  is="acme:Point obix:Point"/>

<obj href="http://acme.com/def/CustomPoint"
  is="http://acme.com/def/Point http://obix.org/def/Point"/>
```


3 oBIX Binary

In addition to the XML encoding, a binary encoding is defined for the oBIX data model. The binary encoding allows oBIX objects to be serialized with higher compression using less computing resources. The use case for binary encoding is targeted for severely constrained edge devices and sensor networks such as 6LoWPANs. When possible, an XML encoding SHOULD always be preferred over a binary encoding.

Full fididelty with oBIX object model is maintained with the binary encoding. All object types and facets are preserved. However XML extensions such as custom namespaces, elements, and attributes are not address by the binary encoding. The oBIX binary encoding is based strictly on the obix data model itself, not its XML InfoSet.

3.1 Binary Overview

The oBIX data model is comprised of 16 object types (elements in XML) and 19 facets (attributes in XML). The oBIX binary encoding is based on assigning a numeric code to each object type and to each facet type. We format these codes using a byte header with the bits structured as:

```
7654 3210
MCCC CCVV
```

The top most bit M is the more flag, it is used to indicate more facets follow. Bits 6 through 2 are used to store a 5-bit numeric code for object types and facet types. The bottom 2 bits are used to indicate a 2-bit numeric code for how the value of the object or facet is encoded.

The binary grammar is defined according to the following BNF productions:

```
<obj>      :=  <objHeader> [objVal] (facet)* [children]
<facet>    :=  <facetHeader> [facetVal]   |
               <facetHeader> <string> <value>
<children> :=  (<obj>)*
```

All documents start with a one byte objHeader structured as a MCCCCCVV bitmask. The 5-bit C mask indicates an Obj Code specified in Binary Constants table. If the object type contains a value encoding (specified in the Obj Value column), then the 2-bit V mask indicates how the following bytes are used to encode the “val” attribute. If the objHeader has the more bit set, then one or more facet productions follow. Facets are encoded with a one byte header using the same MCCCCCVV bitmask, except the 5-bit C mask indicates a Facet Code (not an Obj Code). The facet value is encoded using the 2-bit V mask. If one of the facets includes the hasChildren code, then one or more child objects follow terminated by the endChildren object code.

3.2 Binary Constants

The following table enumerates the Obj Codes and Facet Codes which are encoded into 5-bits in the MCCCCCVV bitmask. The Obj Value and Facet Value columns specifies how to interpret the 2-bit V code for the value encoding.

Numeric Code	Constant	Obj Code	Obj Value	Facet Code	Facet Value
1 << 2	0x04	obj	none	hasChildren	none
2 << 2	0x08	bool	bool	name	str
3 << 2	0x0C	int	int	href	str
4 << 2	0x10	real	real	is	str
5 << 2	0x14	str	str	of	str

6 << 2	0x18	enum	str	in	str
7 << 2	0x1C	uri	str	out	str
8 << 2	0x20	abstime	abstime	null	bool
9 << 2	0x24	reltime	reltime	icon	str
10 << 2	0x28	date	date	displayName	str
11 << 2	0x2C	time	time	display	str
12 << 2	0x30	list	none	writable	bool
13 << 2	0x34	op	none	min	obj specific
14 << 2	0x38	feed	none	max	obj specific
15 << 2	0x3C	ref	none	unit	str
16 << 2	0x40	err	none	precision	int
17 << 2	0x44	childrenEnd	none	range	str
18 << 2	0x48			tz	str
19 << 2	0x4C			status-0	status-0
20 << 2	0x50			status-1	status-1
21 << 2	0x54			customFacet	facet specific

3.3 Value Encodings

Each obj type and facet type may have an associated value encoding. For example, to encode the precision facet we must specify the facet code 0x40 plus the value of that facet which happens to be an integer. The object types bool, int, enum, real, str, uri, abstime, reltime, date, and time are always implied to have their value encoded (equivalent to the val attribute in XML).

3.3.1 Bool Encodings

The following boolean encodings are supported:

Constant	Encoding	Description
0	false	Indicates false value
1	true	Indicates true value

The boolean encodings are fully specified in the 2-bit V mask. No extra bytes are required. Examples:

```
•    <bool val="false"/>  =>  08
•    <bool val="true"/>   =>  09
```

The obj code for bool is 0x08. In the case of false we bit-wise OR this with a value code of 0, so the complete encoding is the single byte 0x08. When val is true, we bitwise OR 0x08 with 0x01 with a result of 0x09.

3.3.2 Int Encodings

The following integer encodings are supported:

Constant	Encoding	Description
0	u1	Unsigned 8-bit integer value
1	u2	Unsigned 16-bit integer value
2	s4	Signed 32-bit integer value
3	s8	Signed 64-bit integer value

Integers between 0 and 255 can be encoded in one byte. Larger numbers require 2, 4, or 8 bytes. Numbers outside of the 64-bit range are not supported. Examples:

```
<int val="34"/>             =>  0C 22
<int val="2093 "/>          =>  0D 08 2D
<int val="76000"/>          =>  0E 00 01 28 E0
<int val="-300"/>           =>  0E FF FF FE D4
<int val="12345678901"/>    =>  0F 00 00 00 02 DF DC 1C 35
```

The obj code for int is 0x0C. In first example, the value can be encoded as an unsigned 8-bit number, so we mask 0x0C with the value code 0x00 and then encode 34 using one byte. The second example is a u2 encoding, so we mask 0x0C with value code 0x01 to get 0x0D and then use two additional bytes to encode 2093 as a 16-bit unsigned integer. The other examples illustrate how values would be encoded in s4 and s8. Encoders SHOULD select the encoding type which results in the fewest number of bytes.

3.3.3 Real Encodings

The following real encodings are supported:

Constant	Encoding	Description
0	f4	32-bit IEEE floating point value
1	f8	64-bit IEEE floating point value

Examples:

```
<real val="75.3"/>        =>  10 42 96 99 9A
<real val="15067.059"/>  =>  11 40 CD 6D 87 8D 4F DF 3B
```

3.3.4 Str Encodings

The following str encodings are supported:

Constant	Encoding	Description
0	utf8	null terminated UTF-8 string
1	prev	u2 index of previously encoded string

String encoding are used for many obj and facet values. Every time a string value is encoded within a given document, it is assigned a zero based index number. The first string encoded as utf8 is assigned zero, the second one, and so on. If subsequent string values have the exact same value, then the prev value encoding is used to reference the previous string via its index number. This requires binary decoders to keep track of all strings during decoding, since later occurrences in the document might reference that string.

Simple example which illustrates a null terminated string:

```
<str val="obix"/>   =>   14 6F 62 69 78 00

```

Complex example which illustrates two strings with the same value:

```
<obj>
  <str val="abc"/>
  <str val="abc"/>
</obj>                    =>  84 04 14 61 62 63 00 15 00 00 44
```


The first byte 0x84 is the obj code masked with the more bit The next byte 0x04 is the hasChildren marker which indicates that children objects follow (covered further in section 3.5). The next byte is the 0x14 str obj code masked with the 0x00 utf8 value code followed by the 61 62 63 00 encoding of “abc”. The next byte 0x15 is the str obj type 0x14 masked with the 0x01 prev value code, followed by the u2 encoding of index zero which references string value zero “abc”. The last byte 0x44 is the end of children marker.

3.3.5 Abstime Encodings

The following abstime encodings are supported:

Constant	Encoding	Description
0	sec	signed 32-bit number of seconds since epoch
1	ns	signed 64-bit number of nanoseconds since epoch

The epoch for oBIX timestamps is defined as midnight 1 January 2000 UTC. Times before the epoch are represented as negative numbers. Encoding with seconds provides a range of +/-68 years. The nanosecond encoding provides a range of +/-292 years. Timestamps outside of this range are not supported. Examples:

```
<abstime val="2000-01-30T00:00:00Z"/>       => 20 00 26 3B 80
<abstime val="1999-12-01T00:00:00Z"/>       => 20 FF D7 21 80
<abstime val="2009-10-20T13:00:00-04:00"/>  => 20 12 70 A9 10
<abstime val="2009-10-20T13:00:00.123Z"/>   => 21 04 4B 10 30 8D 78 F4 C0
```

The first example is encoded as 0x00263B80 which equates to 29x24x60x60 seconds since the oBIX epoch. The second example illustrates a negative number seconds for a timestamp before the epoch. The last example illustrates a 64-bit nanosecond encoding.

3.3.6 Reltime Encodings

The following reltime encodings are supported:

Constant	Encoding	Description
0	sec	signed 32-bit number of seconds
1	ns	signed 64-bit number of nanoseconds

Consistent with the abstime encoding, both a second and nanosecond encoding are provided. No support is provided for ambiguous periods such as 1 month which don’t map to a fixed number of seconds. Examples:

```
<reltime val="PT5M"/>      =>  24 00 00 01 2C
<reltime val="PT0.123S"/>  =>  25 00 00 00 00 07 54 D4 C0
```

3.3.7 Time Encodings

The following time encodings are supported:

Constant	Encoding	Description
0	sec	unsigned 32-bit number of seconds since midnight
1	ns	unsigned 64-bit number of nanoseconds since midnight

The time encoding works similar to reltime using a number of seconds or nanoseconds since midnight. Examples:

```
<time val="04:30:00"/>      =>  2C  00  00  3F  48
<time val="04:30:00.123"/>  =>  2D  00  00  0E  BB  E2  93  A4  C0
```

3.3.8 Date Encodings

The following date encodings are supported:

Constant	Encoding	Description
0	yymd	u2 year, u1 month 1-12, u1 day 1-31

Dates are encoded using four bytes. The year is encoded as a common era year via a 16-bit integer, the month as a 8-bit integer between 1 and 12, and the day as an 8-bit integer between 1 and 31. Examples:

- `<date val="2009-10-20"/>  =>  28  07  D9  0A  14`

3.3.9 Status Encodings

The following status encodings are supported:

Constant	Encoding	Description
0	status-0-disabled	disabled status
1	status-0-fault	fault status
2	status-0-down	down status
3	status-0-unacked-alarm	unackedAlarm status
0	status-1-alarm	alarm status
1	status-1-unacked	unacked status
2	status-1-overridden	overridden status

The status facet is encoded inline to avoid consuming an extra byte. Since there are eight status values, but only 2-bits for the value encoding we use two different facet codes to give us the required range. The ok status is implied by omitting the status facet. Examples:

```
<obj status="ok"/>             =>  04
<obj status="disabled"/>       =>  84 4C   // 0x4C | 0x00
<obj status="fault"/>          =>  84 4D   // 0x4C | 0x01
<obj status="down"/>           =>  84 4E   // 0x4C | 0x02
<obj status="unackedAlarm"/>   =>  84 4F   // 0x4C | 0x03
<obj status="alarm"/>          =>  84 50   // 0x50 | 0x00
<obj status="unacked"/>        =>  84 51   // 0x50 | 0x01
<obj status="overridden"/>     =>  84 52   // 0x50 | 0x02
```

The first example illustrates the ok status, the entire document is encoded with the one byte obj type code of 0x40. The rest of the examples start with 0x84 which represents the obj type code masked with the more bit. Status values from disabled to unackedAlarm use facet code status-0 and from alarm to overridden use facet code status-1. It is illegal for a single object to define both the status-0 and status-1 facet codes.

3.4 Facets

Facets are encoded according to the value type as specified in the Binary Constants section. The min/max facet value types are implied by their containing object which must match the object value with exception of str which uses integers for min/max. Some examples:

```
<list name="foo"/>                      =>  B0 08 66 6F 6F 00
<list name="foo" displayName="Foo"/>    =>  B0 88 66 6F 6F 00 28 46 6F 6F 00
<int val="3" min="0" max="100"/>        =>  8C 03 B4 00 38 64
<obj href="p4.2"/>                      =>  84 0C 70 34 2E 32  00
```

Note that a string of multiple facets is indicated by masking the 0x80 more bit into the object/facet headers.

3.4.1 Custom Facets

The following extension encodings are supported:

Constant	Encoding	Description
0	extension	Facet name encoded as string value object, followed by value object containing value associated with facet.

Custom facets are facets which are not specified by this standard but rather supplied by a particular implementation. Custom facets will include two objects immediately following the header byte: a string object, specifying the name of the facet, and a value object, specifying the value associated with the facet.

Both the string and value objects associated with the facet must provide a value, and neither object may supply additional facets or contain any child objects. Additionally, the value object associated with the facet must be one of the following object types:

- bool
- int
- real
- str
- enum
- uri
- abstime
- reltime
- date
- time

Other types for the value object are not supported.

Examples:

```
<int val="34" my:int=”50”/>        =>  8C 22 54 14 6D 79 3A 69 6E 6F 00 0C 32
<bool val=”false” my:bool=”true”/>        => 88 54 14 6D 79 3A 69 6E 74 00 09
<bool val=”true” my:str=”hi!”/>   => 89 54 14 6D 79 3A 73 74 72 00 14 68 69 21 00
```

3.5 Children

The special facet code hasChildren and the special object code endChildren are used to encode nested children objects. Let’s look at a simple example:

```
<obj> <bool val="false"/>  </obj>  =>  84 04 08 44
```

Let’s examine each byte: the first byte 0x84 is the mask of obj type code 0x04 with the 0x80 more bit indicating a facet follows. The 0x04 facet code indicates the obj has children. The next byte is interpreted as the beginning of a new object, which is the bool object code 0x08. Since the more bit is not set on the bool object, there are no more facets. The next byte is the endChildren object code indicating we’ve reached the end of the children objects for obj. It serves a similar purpose as the end tag in XML.

Technically the hasChildren facet could have additional facets following it by setting the more bit. However, this specification requires that the hasChildren facet is always declared last within a given object’s facet list. This makes it an encoding error to have the more bit set on the hasChildren facet code.

Let’s look a more complicated example with multiple nested children:

```
<list href="xyz">
  <bool val="false"/>
  <obj><int val="255"/></obj>
</list>                         =>  B0 8C 78 79 7A 00 04 08 84 04 0C FF 44 44

<list>               => B0               // 0x80 | 0x30
href="xyz"           => 8C 78 79 7A 00   // 0x80 | 0x0C | 0x00 + x + y + z
hasChildren          => 04
<bool val="false"/>  => 08
<obj>                => 84               // 0x80 | 0x04
hasChildren          => 04
<int val="255"       => 0C FF            // 0x0C | 0x00 + u1 of 255
endChildren </obj>   => 44
endChildren </list>  => 44
```


4 JSON encoding

The Java script object notation is a lightweight, text-based, language-independent data interchange format. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard (ECMA) [RFC4627].

JSON uses two structures for representing information:

- A collection of name/value pairs
- An ordered list of values

In JSON an object is an unordered set of name/value pairs and the encoding of an object starts with a left brace and ends with a right brace. A colon is used to separate the name and the value and a comma separates multiple name/value pairs. The JSON encoding of oBIX is inspired by JSONML, which provides a lossless two-way conversation between JSON and XML. A Java reference implementation can be found here[1].

The following grammar is used to represent oBIX objects:

```
element
= '{' tag-identifier ',' attribute-list ', "nodes":[' element-list ']}'
| '{' tag-identifier ',' attribute-list '}'
| '{' tag-identifier ', "nodes":['element-list ']}'
| '{' tag-identifier '}'
| string
;

tag-identifier
= "tag":tag-name
;

tag-name
= string
;

attribute-list
= attribute ',' attribute-list
| attribute
;

attribute
= attribute-name ':' attribute-value
;

attribute-name
= string
;

attribute-value
= string
| number
```

[1] http://json.org/java/


```
| 'true'
| 'false'
;

element-list
= element ',' element-list
| element
;
```

4.1 Object and value encoding rules

Objects MUST be encoded according to the grammar given above. The oBIX object is encoded as JSON object which an unordered list of name/value pairs. The object type which is used as element name in XML is encoded as a name/value pair using “tag” as name and the object type as string value.

The XML and JSON representation of a simple `obj`:

```
<obj/> → {“tag”:”obj”}
```

The attributes of an object are mapped to name value/pairs:

```
<obj name=”myName” href=”/myHref”> → {“tag”:”obj”, “name”:”myName”, “href”:”/myHref”/>
```

If objects have an extent, the children objects contained in this extend are mapped to a name/value pair using “nodes” as name and an ordered array of objects as value.

The XML representation of an object with extend is mapped to the JSON representation as shown in the examples below.

XML:

```
<obj href=”/a/”>
    <obj name=”b” href=”b”>
            <obj name=”c”/>
            <ref name=”d” href=”d”/>
    </obj>
</obj>
```

JSON:

```
{“tag”:”obj”, “href”:”/a/”, “nodes”:[
    {“tag”,”obj”, “name”:”b”, “href”:”b”}, “nodes”:[
            {“tag”,”obj”, “name”, “c”},
            {“tag”,”ref”, “name”, “d”, “href”:”d”},
            ]
    }]
}
```

4.1.1 Bool encoding

The `xs:boolean` val attribute of the bool object is mapped to the `true` or `false` literals of JSON.

```
<bool val=”true”/> → {“tag”:”bool”, “val”:true}
```

4.1.2 Int encoding

The `xs:long` val attribute of the int object is mapped to the number representation of JSON.

```
<int val=”5”/> → {“tag”:”int”, “val”:5}
```

4.1.3 Real encoding

The `xs:long` val attribute of the int object is mapped to the number representation of JSON.

```
<real val=”5.5”/> → {“tag”:”real”, “val”:5.5}
```

4.1.4 Other types and facets

All other types and facets are mapped to name/value pairs using JSON string representation. Facets are mapped to name/value pairs as described by the rules above.

4.2 XML Namespace

If namespace information should be preserved in the JSON encoding, namespace prefixes SHOULD be normalized before the object is encoded to JSON as shown in the examples below:

Object with namespace prefixes in use:

```
<obj xmlns:acme="http://acme.com/def/" href="acme:CustomPoint"
  is="acme:Point obix:Point"/>
```

Object with normalized namespace information:

```
<obj href="http://acme.com/def/CustomPoint"
  is="http://acme.com/def/Point http://obix.org/def/Point"/>
```

JSON encoded object with normalized namespace information:

```
{tag:”obj”, href:”http://acme.com/def/CustomPoint”, is:”http://acme.com/def/Point
http://obix.org/def/Point”}
```

4.3 Examples

The following examples illustrate the JSON encoding:

Example – oBIX About:

XML:

```
<obj name="about">
     <str name="obixVersion" val="1.1"/>
     <str name="serverName" val="obix"/>
     <abstime name="serverTime" val="2006-02-08T09:40:55.000+05:00:00Z"/>
     <abstime name="serverBootTime" val="2006-02-08T09:33:31.980+05:00:00Z"/>
     <str name="vendorName" val="Acme, Inc."/>
     <uri name="vendorUrl" val="http://www.acme.com"/>
     <str name="productName" val="Acme oBIX Server"/>
     <str name="productVersion" val="1.0.3"/>
     <uri name="productUrl" val="http://www.acme.com/obix"/>
</obj>
```

JSON:

```
{“tag”:”obj”, “name”:”about”, “nodes”:[
    {“tag”:”str”, “name”:”obixVersion”, “val”:”1.1”},
    {“tag”:”str”, “name”:”serverName”, “val”:”obix”},
    {“tag”:”abstime”, “name”:”serverTime”, “val”:”2006-02-08T09:40:55.000+05:00:00Z”},
    {“tag”:”abstime”, “name”:”serverBootTime”, “val”:”2006-02-
08T09:33:31.980+05:00:00Z”},
    {“tag”:”str”,”name”:”vendorName”, “val”:”Acme, Inc.”},
    {“tag”:”uri”,”name”:”vendorURL”, “val”:“http://www.acme.com“},
    {“tag”:”str”,”name”:”productName”, “val”:”Acme oBIX Server”},
    {“tag”:”str”,”name”:”productVersion”,”val”:”1.0.3”},
    {“tag”:”uri”,”name”:”prodctUrl”,”val”:”http://www.acme.com/obix”}
]}
```

4.4 MIME Type

If a client wants to use JSON encoding it MUST use the JSON MIME type `application/json` according to [RFC4627].

5 EXI encoding

The Efficient XML Interchange [EXI] format is a very compact representation for XML which aims at providing high performance and significantly reduced bandwidth requirements for XML based protocols. It uses a grammar driven approach based on entropy encoding which can be used with schema information but also without any schema information.

5.1 EXI options

EXI provides several encoding options that communicating parties need to agree upon in order to ensure interoperability.

If EXI encoding is used for oBIX the following options MUST be used by a client and server implementation.

5.1.1 Alignment options

In contrast to XML EXI is by default bit-packed, which means the information is stored in the most compact representation as possible, regardless of possible byte boundaries. This allows for example to store 8 Boolean values into one single Byte, versus 8 Bytes with a single character representing the value, e.g. ‘T’ or ‘F’. Even worse if a textual representation like ‘true’ or ‘false’ is used, 4 to 5 Bytes are used for representing the Boolean value.

EXI defines 4 options for alignment: `compress`, `preCompress`, `byteAligned` and `bitPacked`.

In order to have the best possible compression for oBIX `bitPacked` alignment MUST be used.

5.1.2 Preservation options

EXI implementation may provide preservation options specifying which type of XML information should be remained in the EXI representation, like comments, programming instructions, document type declarations and namespace.

For oBIX only name space declarations MUST be preserved. Every other non-relevant information MAY be omitted.

5.2 Non-schema-informed EXI

EXI can be used without any schema information about the XML infoset that shall be encoded. This has the advantage that no schema information is required at the decoders site, but comes with the disadvantage of being less efficient and providing only a limited compression for small payloads.

5.3 Schema-informed EXI

Schema-informed EXI allows making the encoding most efficient even for small payload sizes. Within constrained environments schema-informed EXI SHALL be used to in order to have the best compression effect. With object encoders and decoders even the performance penalty of processing XML structures in memory can be avoided.

For schema-informed the normative obix.xsd schema file representing the oBIX 1.1 object model MUST be used in order to provide interoperability among different vendor implementations.

For content negotiation and to determine if schema-informed or non-schema-informed EXI encoding should be used either an out-of-band agreement between a client and server need to be done or the EXI best practices [EXI BP] or the guidelines in [EXI MR] need to be followed.

5.4 MIME types

If a client wants to use EXI encoding it MUST use the MIME type `application/exi` for EXI without schema information and the MIME type `application/x-obix-exi` for schema-informed representation.

6 Conformance

An implementation is compliant with this specification if it implements all MUST or REQUIRED level requirements. An implementation MUST specify its supported encodings.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Ron Ambrosio, IBM
Brad Benson, Trane
Ron Bernstein, LonMark International*
Ludo Bertsch, Continental Automated Buildings Association (CABA)
Chris Bogen, US Department of Defense
Rich Blomseth, Echelon Corporation
Anto Budiardjo, Clasma Events, Inc.
Jochen Burkhardt, IBM
JungIn Choi, Kyungwon University
David Clute, Cisco Systems, Inc.*
Toby Considine, University of North Carolina at Chapel Hill
William Cox, Individual
Robert Dolin, Echelon Corporation
Marek Dziedzic, Treasury Board of Canada, Secretariat
Brian Frank, SkyFoundry
Craig Gemmill, Tridium, Inc.
Matthew Giannini, Tridium, Inc.
Harald Hofstätter, Institute of Computer Aided Automation, Vienna University of Technology
Markus Jung, Institute of Computer Aided Automation, Vienna University of Technology
Christopher Kelly, Cisco Systems
Wonsuk Ko, Kyungwon University
Perry Krol, TIBCO Software Inc.
Corey Leong, Individual
Ulf Magnusson, Schneider Electric
Brian Meyers, Trane
Jeremy Roberts, LonMark International
Thorsten Roggendorf, Echelon Corporation
Anno Scholten, Individual
John Sublett, Tridium, Inc.
Dave Uden, Trane
Ron Zimmer, Continental Automated Buildings Association (CABA)*
Robert Zach, Institute of Computer Aided Automation
Rob Zivney, Hirsch Electronics Corporation
Markus Jung, Vienna University of Technology

Appendix B. Revision History

Revision	Date	Editor	Changes Made
wd01	26 Mar 13	Markus Jung	Initial creation with XML and Binary encoding taken from the oBIX 1.1 WD07 working draft.
wd02	24 Apr 13	Markus Jung	First draft JSON and EXI encoding.
wd03	22 May 13	Markus Jung	Added JSON section on handling XML namespaces, shorter JSON names.
wd04	13 Jun 13	Markus Jung	Refined the use of examples (normative/non normative), EXI content negotiation.
wd05	28 Jun 13	Markus Jung	Updated reference section
Wd06	8 Jul 13	Toby Considine	Updated acknowledgements

