OBIX Version 1.1

Working Draft 06

08 June 2010

Specification URIs:

This Version:

http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.html
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.doc
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.pdf

Previous Version:

http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-04.html
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-04.doc
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-04.pdf

Latest Version:

http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.html
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.doc
http://docs.oasis-open.org/oBIX/v1.1/oBIX-1.0-spec-wd-05.pdf

Technical Committee:

OASIS Open Building Information Exchange TC

Chair(s):

Toby Considine

Editor(s):

Brian Frank

Related work:

This specification replaces or supersedes:

OASIS oBIX Committee Specification 1.0

This specification is related to:

OASIS Specification WS-Calendar V1.1, in process

Declared XML Namespace(s):

http://docs.oasis-open.org/oBIX/2010interim

Abstract:

This document specifies an object model and XML format used for machine-to-machine (M2M) communication

Status:

This document was last revised or approved by the Open Building Information Exchange Exchange Technical Committee on the above date. The level of approval is also listed above. Check the “Latest Version” or “Latest Approved 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.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/oBIX/

Notices

Table of Contents

1 Introduction 8
1.1 Design Concerns 8
1.1.1 XML 8
1.1.2 Networking 8
1.1.3 Normalization 8
1.1.4 Foundation 8
1.2 Terminology 9
1.3 Normative References 9
1.4 Non-Normative References 9
2 Quick Start 11
3 Architecture 13
3.1 Object Model 13
3.2 XML 13
3.3 URIs 14
3.4 REST 14
3.5 Contracts 14
3.6 Extendibility 15
4 Object Model 16
4.1 obj 16
4.2 bool 17
4.3 int 17
4.4 real 17
4.5 str 17
4.6 enum 17
4.7 abstime 18
4.8 reltime 18
4.9 date 18
4.10 time 18
4.11 uri 19
4.12 list 19
4.13 ref 19
4.14 err 19
4.15 op 19
4.16 feed 19
4.17 Null 20
4.18 Facets 20
4.18.1 displayName 20
4.18.2 display 20
4.18.3 icon 20
4.18.4 min 20
4.18.5 max 21
4.18.6 precision 21
4.18.7 range 21

4.18.8 status 21
4.18.9 tz 22
4.18.10 unit 22
4.18.11 writable 22
5 Naming 24
5.1 Name 24
5.2 Href 24
5.3 HTTP Relative URIs 24
5.4 Fragment URIs 25
6 Contracts 26
6.1 Contract Terminology 26
6.2 Contract List 26
6.3 Is Attribute 27
6.4 Contract Inheritance 27
6.5 Override Rules 28
6.6 Multiple Inheritance 28
6.6.1 Flattening 28
6.6.2 Mixins 29
6.7 Contract Compatibility 30
6.8 Lists (and Feeds) 30
7 XML Encoding 31
7.1 Design Philosophy 31
7.2 XML Syntax 31
7.3 XML Encoding 31
7.4 XML Decoding 32
7.5 XML Namespace 32
7.6 Namespace Prefixes in Contract Lists 32
8 Binary Encoding 33
8.1 Binary Overview 33
8.2 Binary Constants 33
8.3 Value Encodings 34
8.3.1 Bool Encodings 34
8.3.2 Int Encodings 35
8.3.3 Real Encodings 35
8.3.4 Str Encodings 35
8.3.5 Abstime Encodings 36
8.3.6 Reltime Encodings 36
8.3.7 Time Encodings 36
8.3.8 Date Encodings 37
8.3.9 Status Encodings 37
8.4 Facets 37
8.4.1 Custom Facets 38
8.5 Children 38
9 Operations 40
10 Object Composition 41

10.1 Containment 41
10.2 References 41
10.3 Extents 41
10.4 XML 42
11 Networking 43
11.1 Request / Response 43
11.1.1 Read 43
11.1.2 Write 43
11.1.3 Invoke 44
11.2 Errors 44
11.3 Lobby 44
11.4 About 44
11.5 Batch 45
12 Core Contract Library 47
12.1 Nil 47
12.2 Range 47
12.3 Weekday 47
12.4 Month 47
12.5 Units 48
13 Watches 50
13.1 WatchService 50
13.2 Watch 50
13.2.1 Watch.add 51
13.2.2 Watch.remove 51
13.2.3 Watch.pollChanges 51
13.2.4 Watch.pollRefresh 52
13.2.5 Watch.lease 52
13.2.6 Watch.delete 52
13.3 Watch Depth 52
13.4 Feeds 52
14 Points 54
14.1 Writable Points 54
15 History 55
15.1 History Object 55
15.2 History Queries 56
15.2.1 HistoryFilter 56
15.2.2 HistoryQueryOut 56
15.2.3 HistoryRecord 56
15.2.4 History Query Example 56
15.3 History Rollups 57
15.3.1 HistoryRollupIn 57
15.3.2 HistoryRollupOut 57
15.3.3 HistoryRollupRecord 57
15.3.4 Rollup Calculation 58
15.4 History Feeds 59

15.5 History Append 59
15.5.1 HistoryAppendIn 59
15.5.2 HistoryAppendOut 59
16 Alarming 61
16.1 Alarm States 61
16.1.1 Alarm Source 61
16.1.2 StatefulAlarm and AckAlarm 62
16.2 Alarm Contracts 62
16.2.1 Alarm 62
16.2.2 StatefulAlarm 62
16.2.3 AckAlarm 62
16.2.4 PointAlarms 63
16.3 AlarmSubject 63
16.4 Alarm Feed Example 63
17 Security 65
17.1 Error Handling 65
17.2 Permission based Degradation 65
18 HTTP Binding 66
18.1 Requests 66
18.2 MIME Type 66
18.3 Content Negotiation 66
18.4 Security 66
18.5 Localization 67
19 SOAP Binding 68
19.1 SOAP Example 68
19.2 Error Handling 68
19.3 Security 68
19.4 Localization 68
19.5 WSDL 69
20 Conformance 71
Acknowledgements 72
A. Appendices 73
Revision History 74

1 Introduction

oBIX is designed to provide access to the embedded software systems which sense and control the world around us. Historically integrating to these systems required custom low level protocols, often custom physical network interfaces. But now the rapid increase in ubiquitous networking and the availability of powerful microprocessors for low cost embedded devices is weaving these systems into the very fabric of the Internet. Generically the term M2M for Machine-to-Machine describes the transformation occurring in this space because it opens a new chapter in the development of the Web - machines autonomously communicating with each other. The oBIX specification lays the groundwork building this M2M Web using standard, enterprise friendly technologies like XML, HTTP, and URIs.

1.1 Design Concerns

The following design points illustrate the problem space oBIX attempts to solve:

- **XML**: representing M2M information in a standard XML syntax;
- **Networking**: transferring M2M information in XML over the network;
- **Normalization**: standard representations for common M2M features: points, histories, and alarms;
- **Foundation**: providing a common kernel for new standards;

1.1.1 XML

The principal requirement of oBIX is to develop a common XML syntax for representing information from diverse M2M systems. The design philosophy of oBIX is based on a small, but extensible data model which maps to a simple fixed XML syntax. This core object model and its XML syntax is simple enough to capture entirely in one illustration provided in Chapter 4. The object model's extensibility allows for the definition of new abstractions through a concept called *contracts*. The majority of the oBIX specification is actually defined in oBIX itself through contracts.

1.1.2 Networking

Once we have a way to represent M2M information in XML, the next step is to provide standard mechanisms to transfer it over networks for publication and consumption. oBIX breaks networking into two pieces: an abstract request/response model and a series of protocol bindings which implement that model. Version 1.1 of oBIX defines two protocol bindings designed to leverage existing web service infrastructure: an HTTP REST binding and a SOAP binding.

1.1.3 Normalization

There are a few concepts which have broad applicability in systems which sense and control the physical world. Version 1.1 of oBIX provides a normalized representation for three of these:

- **Points**: representing a single scalar value and it's status – typically these map to sensors, actuators, or configuration variables like a setpoint;
- **Histories**: modeling and querying of time sampled point data. Typically edge devices collect a time stamped history of point values which can be fed into higher level applications for analysis;
- **Alarming**: modeling, routing, and acknowledgment of alarms. Alarms indicate a condition which requires notification of either a user or another application.

1.1.4 Foundation

The requirements and vertical problem domains for M2M systems are immensely broad – too broad to cover in one single specification. oBIX is deliberately designed as a fairly low level specification, but with a powerful extension mechanism based on contracts. The goal of oBIX is to lay the groundwork for a common object model and XML syntax which serves as the foundation for new specifications. It is hoped that a stack of specifications for vertical domains can be built upon oBIX as a common foundation.

1.2 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 **Error! Reference source not found.**.

1.3 Normative References

[RFC2119] **S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.**

RFC2246 T. Dierks, C. Allen *Transport Layer Security (TLS) Protocol Version 1.0*, http://www.ietf.org/rfc/rfc2246.txt, IETF RFC 2246, January 1999.

SOA-RM OASIS Standard, *Reference Model for Service Oriented Architecture 1.0*, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf [WS-Calendar] **OASIS WS-Calendar Technical Committee, specification in progress**

[CalWS] **CalConnect TC-XML , specification in progress**

[CEFACT] [UML] ***Unified Modeling Language (UML), Version 2.2*, Object Management Group, February, 2009. http://www.omg.org/technology/documents/formal/uml.htm .**

[XLINK] ***XML Linking Language (XLink) Version 1.1. S DeRose, E Maler, D Orchard, N Walsh, http://www.w3.org/TR/xlink11/ May 2010.***

xCal C. Daboo, M Douglas, S Lees *xCal: The XML format for iCalendar,* http://tools.ietf.org/html/draft-daboo-et-al-icalendar-in-xml-03, Internet-Draft, April 2010.

Calendar Resource Schema C. Joy, C. Daboo, M Douglas, *Schema for representing resources for calendaring and scheduling services*, http://tools.ietf.org/html/draft-cal-resource-schema-00, (Internet-Draft), April 2010.

CalWS CalConnect draft in Process.

.

XLINK S DeRose, E Maler, D Orchard, N Walsh *XML Linking Language (XLink) Version 1.1*., http://www.w3.org/TR/xlink11/ May 2010.

XPOINTER S DeRose, E Maler, R Daniel Jr. *XPointer xpointer Scheme*, http://www.w3.org/TR/xptr-xpointer/ December 2002.

XML Schema PV Biron, A Malhotra, *XML Schema Part 2: Datatypes Second Edition*, http://www.w3.org/TR/xmlschema-2/ October 2004.

1.4 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

SOAP M Gudgin, M Hadley, N Mendelsohn, J Moreau, H Nielsen, A Karmarkar, Y Lafon, W3C SOAP Version 1.2 (Second Edition), W3C Recommendation 27 April 2007

2 Quick Start

This chapter is for those eager beavers who want to immediately jump right into oBIX and all its angle bracket glory. The best way to begin is to take a simple example that anybody is familiar with – the staid thermostat. Let's assume we have a very simple thermostat. It has a temperature sensor which reports the current space temperature and it has a setpoint that stores the desired temperature. Let's assume our thermostat only supports a heating mode, so it has a variable that reports if the furnace should currently be on. Let's take a look at what our thermostat might look like in oBIX XML:

```
<obj href="http://myhome/thermostat">
  <real name="spaceTemp" unit="obix:units/fahrenheit" val="67.2"/>
  <real name="setpoint" unit="obix:units/fahrenheit" val="72.0"/>
  <bool name="furnaceOn" val="true"/>
</obj>
```

The first thing to notice is that there are three element types. In oBIX there is a one-to-one mapping between *objects* and *elements*. Objects are the fundamental abstraction used by the oBIX data model. Elements are how those objects are expressed in XML syntax. This document uses the term object and sub-objects, although you can substitute the term element and sub-element when talking about the XML representation.

The root `obj` element models the entire thermostat. Its `href` attribute identifies the URI for this oBIX document. There are three child objects for each of the thermostat's variables. The `real` objects store our two floating point values: space temperature and setpoint. The `bool` object stores a boolean variable for furnace state. Each sub-element contains a `name` attribute which defines the role within the parent. Each sub-element also contains a `val` attribute for the current value. Lastly we see that we have annotated the temperatures with an attribute called `unit` so we know they are in Fahrenheit, not Celsius (which would be one hot room). The oBIX specification defines a bunch of these annotations which are called *facets*.

In real life, sensor and actuator variables (called *points*) imply more semantics than a simple scalar value. In other cases such as alarms, it is desirable to standardize a complex data structure. oBIX captures these concepts into *contracts*. Contracts allow us to tag objects with normalized semantics and structure.

Let's suppose our thermostat's sensor is reading a value of -412°F? Clearly our thermostat is busted, so we should report a fault condition. Let's rewrite the XML to include the status facet and to provide additional semantics using contracts:

```
<obj href="http://myhome/thermostat/">

  <!-- spaceTemp point -->
  <real name="spaceTemp" is="obix:Point"
        val="-412.0" status="fault"
        unit="obix:units/fahrenheit"/>

  <!-- setpoint point -->
  <real name="setpoint" is="obix:Point"
        val="72.0"
        unit="obix:units/fahrenheit"/>

  <!-- furnaceOn point -->
  <bool name="furnaceOn" is="obix:Point" val="true"/>

</obj>
```

Notice that each of our three scalar values are tagged as `obix:Points` via the `is` attribute. This is a standard contract defined by oBIX for representing normalized point information. By implementing these contracts, clients immediately know to semantically treat these objects as points.

Contracts play a pivotal role in oBIX for building new abstractions upon the core object model. Contracts are slick because they are just normal objects defined using standard oBIX syntax (see Chapter 14 to take a sneak peek at the point contracts).

3 Architecture

The oBIX architecture is based on the following principles:

- **Object Model**: a concise object model used to define all oBIX information.
- **XML Encoding**: a simple XML syntax for expressing the object model.
- **Binary Encoding**: a simple binary encoding for constrained devices and networks such as 6LoWPAN sensor networks
- **URIs**: URIs are used to identify information within the object model.
- **REST**: a small set of verbs is used to access objects via their URIs and transfer their state via XML.
- **Contracts**: a template model for expressing new oBIX “types”.
- **Extendibility**: providing for consistent extendibility using only these concepts.

3.1 Object Model

All information in oBIX is represented using a small, fixed set of primitives. The base abstraction for these primitives is cleverly called *object*. An object can be assigned a URI and all objects can contain other objects.

There are ten special kinds of *value objects* used to store a piece of simple information:

- *bool*: stores a boolean value - true or false;
- *int*: stores an integer value;
- *real*: stores a floating point value;
- *str*: stores a UNICODE string;
- *enum*: stores an enumerated value within a fixed range;
- *abstime*: stores an absolute time value (timestamp);
- *reltime*: stores a relative time value (duration or time span);
- *date*: stores a specific date as day, month, and year;
- *time*: stores a time of day as hour, minutes, and seconds;
- *uri*: stores a Universal Resource Identifier;

Note that any value object can also contain sub-objects. There are also a couple of other special object types: *list*, *op, feed, ref* and *err*.

3.2 XML

oBIX is all about a simple XML syntax to represent its underlying object model. Each of the object types map to one type of element. The value objects represent their data value using the `val` attribute. All other aggregation is simply nesting of elements. A simple example to illustrate:

```
<obj href="http://bradybunch/people/Mike-Brady/">
  <obj name="fullName">
    <str name="first" val="Mike"/>
    <str name="last" val="Brady"/>
  </obj>
  <int name ="age" val="45"/>
  <ref name="spouse" href="/people/Carol-Brady"/>
  <list name="children">
    <ref href="/people/Greg-Brady"/>
    <ref href="/people/Peter-Brady"/>
    <ref href="/people/Bobby-Brady"/>
    <ref href="/people/Marsha-Brady"/>
```

```
    <ref href="/people/Jan-Brady"/>
    <ref href="/people/Cindy-Brady"/>
  </list>
</obj>
```

Note in this simple example how the `href` attribute specifies URI references which may be used to fetch more information about the object. Names and hrefs are discussed in detail in Chapter 5.

3.3 URIs

No architecture is complete without some sort of naming system. In oBIX everything is an object, so we need a way to name objects. Since oBIX is really about making information available over the web using XML, it makes to sense to leverage the venerable URI (Uniform Resource Identifier). URIs are the standard way to identify “resources” on the web.

Often URIs also provide information about how to fetch their resource - that’s why they are often called URLs (Uniform Resource Locator). From a practical perspective if a vendor uses HTTP URIs to identify their objects, you can most likely just do a simple HTTP GET to fetch the oBIX document for that object. But technically, fetching the contents of a URI is a protocol binding issue discussed in later chapters.

The value of URIs are that they come with all sorts of nifty rules already defined for us (see RFC 3986). For example URIs define which characters are legal and which are illegal. Of great value to oBIX is *URI references* which define a standard way to express and normalize relative URIs. Plus most programming environments have libraries to manage URIs so developers don’t have to worry about nitty gritty normalization details.

3.4 REST

Many savvy readers may be thinking that objects identified with URIs and passed around as XML documents is starting to sound a lot like REST – and you would be correct. REST stands for REpresentational State Transfer and is an architectural style for web services that mimics how the World Wide Web works. The WWW is basically a big web of HTML documents all hyperlinked together using URIs. Likewise, oBIX is basically a big web of XML object documents hyperlinked together using URIs.

REST is really more of a design style, than a specification. REST is resource centric as opposed to method centric - resources being oBIX objects. The methods actually used tend to be a very small fixed set of verbs used to work generically with all resources. In oBIX all network requests boil down to three request types:

- **Read**: an object
- **Write**: an object
- **Invoke**: an operation

3.5 Contracts

In every software domain, patterns start to emerge where many different object instances share common characteristics. For example in most systems that model people, each person probably has a name, address, and phone number. In vertical domains we may attach domain specific information to each person. For example an access control system might associate a badge number with each person.

In object oriented systems we capture these patterns into classes. In relational databases we map them into tables with typed columns. In oBIX we capture these patterns using a concept called *contracts*, which are standard oBIX objects used as a template. Contracts are more nimble and flexible than strongly typed schema languages, without the overhead of introducing new syntax. A contract document is parsed just like any other oBIX document. In geek speak contracts are a combination of prototype based inheritance and mixins.

Why do we care about trying to capture these patterns? The most important use of contracts is by the oBIX specification itself to define new standard abstractions. It is just as important for everyone to agree on normalized semantics as it is as on syntax. Contracts also provide the definitions needed to map to the OO guy's classes or the relational database guy's tables.

3.6 Extendibility

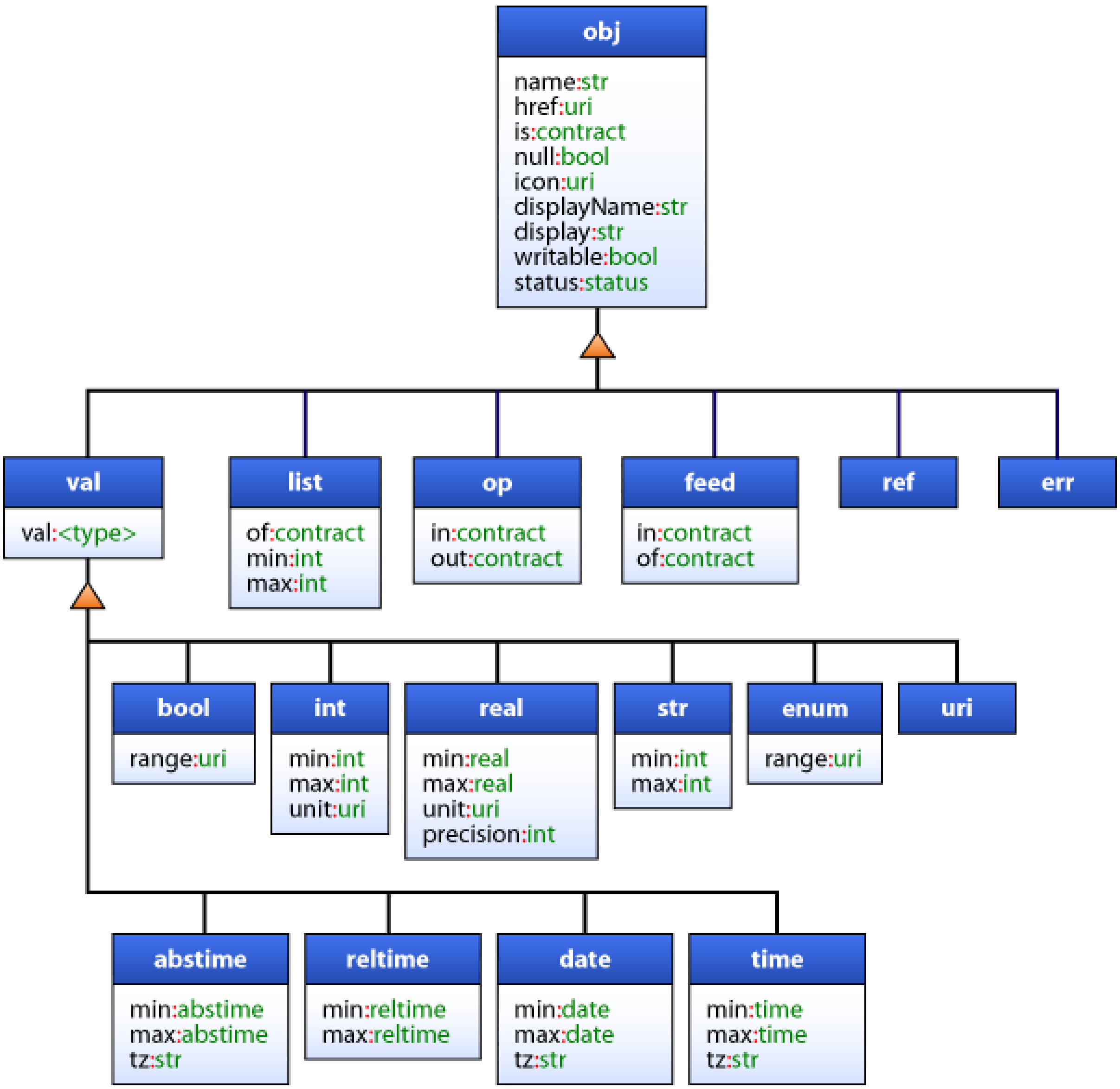
We want to use oBIX as a foundation for developing new abstractions in vertical domains. We also want to provide extendibility for vendors who implement oBIX across legacy systems and new product lines. Additionally, it is common for a device to ship as a blank slate and be completely programmed in the field. This leaves us with a mix of standards based, vendor based, and even project based extensions.

The principle behind oBIX extendibility is that anything new is defined strictly in terms of objects, URIs, and contracts. To put it another way - new abstractions don't introduce any new XML syntax or functionality that client code is forced to care about. New abstractions are always modeled as standard trees of oBIX objects, just with different semantics. That doesn't mean that higher level application code never changes to deal with new abstractions, but the core stack that deals with networking and parsing shouldn't have to change.

This extendibility model is similar to most mainstream programming languages such as Java or C#. The syntax of the core language is fixed with a built in mechanism to define new abstractions. Extendibility is achieved by defining new class libraries using the language's fixed syntax. This means I don't have to update the compiler every time some one adds a new class.

4 Object Model

The oBIX specification is based on a small, fixed set of object types. These object types map one to one to an XML element type. The oBIX object model is summarized in the following illustration. Each box represents a specific object type (and XML element name). Each object type also lists its supported attributes.



4.1 obj

The root abstraction in oBIX is *object*, modeled in XML via the `obj` element. Every XML element in oBIX is a derivative of the `obj` element. Any `obj` element or its derivatives can contain other `obj` elements. The attributes supported on the `obj` element include:

- **name**: defines the object's purpose in its parent object (discussed in the Chapter 5);

- **href**: provides a URI reference for identifying the object (discussed in the Chapter 5);
- **is**: defines the contracts the object implements (discussed in Chapter 6);
- **null**: support for null objects (discussed in Section 4.17)
- **facets**: a set of attributes used to provide meta-data about the object (discussed in Section 4.18);
- **val**: an attribute used only with value objects (`bool`, `int`, `real`, `str`, `enum`, `abstime`, `reltime`, `date`, `time` and `uri`) to store the actual value. The literal representation of values map to XML Schema Part 2: Datatypes - indicated in the following sections via the "`xs:`" prefix.

The contract definition of `obj`:

```
<obj href="obix:obj" null="false" writable="false" status="ok" />
```

4.2 bool

The `bool` object represents a boolean condition of either true or false. Its `val` attribute maps to `xs:boolean` defaulting to false. The literal value of a `bool` MUST be "true" or "false" (the literals "1" and "0" are not allowed). The contract definition:

```
<bool href="obix:bool" is="obix:obj" val="false" null="false"/>
```

An example:

```
<bool val="true"/>
```

4.3 int

The `int` type represents an integer number. Its `val` attribute maps to `xs:long` as a 64-bit integer with a default of 0. The contract definition:

```
<int href="obix:int" is="obix:obj" val="0" null="false"/>
```

An example:

```
<int val="52"/>
```

4.4 real

The `real` type represents a floating point number. Its `val` attribute maps to `xs:double` as a IEEE 64-bit floating point number with a default of 0. The contract definition:

```
<real href="obix:real" is="obix:obj" val="0" null="false"/>
```

An example:

```
<real val="41.06"/>
```

4.5 str

The `str` type represents a string of Unicode characters. Its `val` attribute maps to `xs:string` with a default of the empty string. The contract definition:

```
<str href="obix:str" is="obix:obj" val="" null="false"/>
```

An example:

```
<str val="hello world"/>
```

4.6 enum

The `enum` type is used to represent a value which must match a finite set of values. The finite value set is called the *range*. The `val` attribute of an enum is represented as a string key using `xs:string`. `Enums` default to null. The range of an `enum` is declared via facets using the `range` attribute. The contract definition:

```
<enum href="obix:enum" is="obix:obj" val="" null="true"/>
```

An example:

```
<enum range="/enums/OffSlowFast" val="slow"/>
```

4.7 abstime

The `abstime` type is used to represent an absolute point in time. Its `val` attribute maps to `xs:dateTime`, with the exception that the timezone is required. According to XML Schema Part 2 section 3.2.7.1, the lexical space for abstime is:

```
'-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)
```

Abstimes default to null. The contract definition:

```
<abstime href="obix:abstime" is="obix:obj" val="1970-01-01T00:00:00Z" null="true"/>
```

An example for 9 March 2005 at 1:30PM GMT:

```
<abstime val="2005-03-09T13:30:00Z"/>
```

The timezone offset is required, so the abstime can be used to uniquely relate the abstime to UTC. The optional `tz` facet is used to specify the timezone as a zoneinfo identifier. This provides additional context about the timezone, if available. The timezone offset of the `val` attribute MUST match the offset for the timezone specified by the tz facet, if it is also used. See the `tz` facet section for more information.

4.8 reltime

The `reltime` type is used to represent a relative duration of time. Its `val` attribute maps to `xs:duration` with a default of 0sec. The contract definition:

```
<reltime href="obix:reltime" is="obix:obj" val="PT0S" null="false"/>
```

An example of 15 seconds:

```
<reltime val="PT15S"/>
```

4.9 date

The `date` type is used to represent a day in time as a day, month, and year. Its `val` attribute maps to `xs:date`. According to XML Schema Part 2 section 3.2.9.1, the lexical space for `date` is:

```
'-'? yyyy '-' mm '-' dd
```

Date values in oBIX MUST omit the timezone offset and MUST NOT use the trailing “Z”. Only the tz attribute SHOULD be used to associate the date with a timezone. Date objects default to null. The contract definition:

```
<date href="obix:date" is="obix:obj" val="1970-01-01" null="true"/>
```

An example for 26 November 2007:

```
<date val="2007-11-26"/>
```

The `tz` facet is used to specify the timezone as a zoneinfo identifier. Even when using a timezone, the `val` attribute MUST omit the timezone offset. See the `tz` facet section for more information.

4.10 time

The `time` type is used to represent a time of day in hours, minutes, and seconds. Its `val` attribute maps to `xs:time`. According to XML Schema Part 2 section 3.2.8, the lexical space for `time` is the left truncated representation of `xs:dateTime`:

```
hh ':' mm ':' ss ('.' s+)?
```

Time values in oBIX MUST omit the timezone offset and MUST NOT use the trailing “Z”. Only the tz attribute SHOULD be used to associate the time with a timezone. Time objects default to null. The contract definition:

```
<time href="obix:time" is="obix:obj" val="00:00:00Z" null="true"/>
```

An example for 4:15 AM:

```
<time val="04:15:00"/>
```

The `tz` facet is used to specify the timezone as a zoneinfo identifier. Even when using a timezone, the `val` attribute MUST omit the timezone offset. See the `tz` facet section for more information.

4.11 uri

The `uri` type is used to store a URI reference. Unlike a plain old `str`, a `uri` has a restricted lexical space as defined by RFC 3986 and XML Schema `xs:anyURI` type. Most URIs will also be a URL, meaning that they identify a resource and how to retrieve it (typically via HTTP). The contract:

```
<uri href="obix:uri" is="obix:obj" val="" null="false"/>
```

An example for the oBIX home page:

```
<uri val="http://obix.org/" />
```

4.12 list

The `list` object is a specialized object type for storing a list of other objects. The primary advantage of using a `list` versus a generic `obj` is that lists can specify a common contract for their contents using the `of` attribute. If specified the `of` attribute MUST be a list of URIs formatted as a contract list. The definition of list is:

```
<list href="obix:list" is="obix:obj" of="obix:obj"/>
```

An example list of strings:

```
<list of="obix:str">
  <str val="one"/>
  <str val="two"/>
</list>
```

Lists are discussed in greater detail along with contracts in section 6.8.

4.13 ref

The `ref` object is used to create an out of document reference to another oBIX object. It is the oBIX equivalent of the HTML anchor tag. The contract definition:

```
<ref href="obix:ref " is="obix:obj"/>
```

A `ref` element MUST always specify a `href` attribute. References are discussed in detail in section 10.2.

4.14 err

The `err` object is a special object used to indicate an error. Its actual semantics are context dependent. Typically `err` objects SHOULD include a human readable description of the problem via the `display` attribute. The contract definition:

```
<err href="obix:err" is="obix:obj"/>
```

4.15 op

The `op` object is used to define an operation. All operations take one input object as a parameter, and return one object as an output. The input and output contracts are defined via the `in` and `out` attributes. The contract definition:

```
<op href="obix:op" is="obix:obj" in="obix:Nil" out="obix:Nil"/>
```

Operations are discussed in detail in Chapter 8.

4.16 feed

The `feed` object is used to define a topic for a feed of events. Feeds are used with watches to subscribe to a stream of events such as alarms. A feed SHOULD specify the event type it fires via the `of` attribute. The `in` attribute can be used to pass an input argument when subscribing to the feed (a filter for example).

```
<feed href="obix:feed" is="obix:obj" in="obix:Nil" of="obix:obj"/>
```

Feeds are subscribed via Watches discussed in Chapter 13.

4.17 Null

All objects support the concept of *null*. Null is the absence of a value. Null is indicated using the `null` attribute with a boolean value. All objects default null to false with the exception of `enum` and `abstime` (since any other default would be confusing).

Null is inherited from contracts a little differently than other attributes. See Section 6.4 for details.

4.18 Facets

All objects can be annotated with a predefined set of attributes called *facets*. Facets provide additional meta-data about the object. The set of available facets is: `displayName`, `display`, `icon`, `min`, `max`, `precision`, `range`, and `unit`. Vendors which wish to annotate objects with additional facets should consider using XML namespace qualified attributes.

4.18.1 displayName

The `displayName` facet provides a localized human readable name of the object stored as a `xs:string`:

```
<obj name="spaceTemp" displayName="Space Temperature"/>
```

Typically the `displayName` facet SHOULD be a localized form of the `name` attribute. There are no restrictions on `displayName` overrides from the contract (although it SHOULD be uncommon since `displayName` is just a human friendly version of `name`).

4.18.2 display

The `display` facet provides a localized human readable description of the object stored as a `xs:string`:

```
<bool name="occupied" val="false" display="Unoccupied"/>
```

There are no restrictions on `display` overrides from the contract.

The `display` attribute serves the same purpose as Object.toString() in Java or C#. It provides a general way to specify a string representation for all objects. In the case of value objects (like `bool` or `int`) it SHOULD provide a localized, formatted representation of the `val` attribute.

4.18.3 icon

The `icon` facet provides a URI reference to a graphical icon which may be used to represent the object in an user agent:

```
<object icon="/icons/equipment.png"/>
```

The contents of the `icon` attribute MUST be a URI to an image file. The image file is preferably a 16x16 PNG file. There are no restrictions on `icon` overrides from the contract.

4.18.4 min

The `min` facet is used to define an inclusive minimum value:

```
<int min="5" val="6"/>
```

The contents of the `min` attribute MUST match its associated `val` type. The `min` facet is used with `int`, `real` , `abstime`, `date`, `time`, and `reltime` to define an inclusive lower limit of the value space. It is used with `str` to indicate the minimum number of Unicode characters of the string. It is used with `list` to indicate the minimum number of child objects (named or unnamed). Overrides of the `min` facet may only narrow the value space using a larger value. The `min` facet MUST never be greater than the `max` facet (although they can be equal).

4.18.5 max

The `max` facet is used to define an inclusive maximum value:

```
<real max="70" val="65"/>
```

The contents of the `max` attribute MUST match its associated `val` type. The `max` facet is used with `int`, `real`, `abstime`, `date`, `time`, and `reltime` to define an inclusive upper limit of the value space. It is used with `str` to indicate the maximum number of Unicode characters of the string. It is used with `list` to indicate the maximum number of child objects (named or unnamed). Overrides of the `max` facet may only narrow the value space using a smaller value. The `max` facet MUST never be less than the `min` facet (although they MAY be equal).

4.18.6 precision

The `precision` facet is used to describe the number of decimal places to use for a `real` value:

```
<real precision="2" val="75.04"/>
```

The contents of the `precision` attribute MUST be `xs:int`. The value of the `precision` attribute equates to the number of meaningful decimal places. In the example above, the value of 2 indicates two meaningful decimal places: “75.04”. Typically precision is used by client applications which do their own formatting of `real` values. There are no restrictions on `precision` overrides.

4.18.7 range

The `range` facet is used to define the value space of an enumeration. A `range` attribute is a URI reference to an `obix:Range` object (see section 12.2 for the definition). It is used with the `bool` and `enum` object types:

```
<enum range="/enums/OffSlowFast" val="slow"/>
```

The override rule for `range` is that the specified range MUST inherit from the contract’s range. Enumerations are funny beasts in that specialization of an enum usually involves adding new items to the range. Technically this is widening the enum’s value space, rather than narrowing it. But in practice, adding items into the range is what we desire.

4.18.8 status

The `status` facet is used to annotate an object about the quality and state of the information:

```
<real val="67.2" status="alarm"/>
```

Status is an enumerated string value with one of the following values (ordered by priority):

- **`disabled`**: This state indicates that the object has been disabled from normal operation (out of service). In the case of operations and feeds, this state is used to disable support for the operation or feed.
- **`fault`**: The `fault` state indicates that the data is invalid or unavailable due to a failure condition - data which is out of date, configuration problems, software failures, or hardware failures. Failures involving communications should use the `down` state.
- **`down:`** The `down` state indicates a communication failure.
- **`unackedAlarm:`** The `unackedAlarm` state indicates there is an existing alarm condition which has not been acknowledged by a user – it is the combination of the `alarm` and `unacked` states. The difference between `alarm` and `unackedAlarm` is that `alarm` implies that a user has already acknowledged the alarm or that no human acknowledgement is necessary for the alarm condition. The difference between `unackedAlarm` and `unacked` is that the object has returned to a normal state.

- **`alarm:`** This state indicates the object is currently in the alarm state. The alarm state typically means that an object is operating outside of its normal boundaries. In the case of an analog point this might mean that the current value is either above or below its configured limits. Or it might mean that a digital sensor has transitioned to an undesired state. See Alarming (Chapter 16) for additional information.
- **`unacked:`** The `unacked` state is used to indicate a past alarm condition which remains unacknowledged.
- **`overridden`**: The overridden state means the data is ok, but that a local override is currently in effect. An example of an override might be the temporary override of a setpoint from it's normal scheduled setpoint.
- **`ok:`** The ok state indicates normal status. This is the assumed default state for all objects.

Status MUST be one of the enumerated strings above. It might be possible in the native system to exhibit multiple status states simultaneously, however when mapping to oBIX the highest priority status SHOULD be chosen – priorities are ranked from top (disabled) to bottom (ok).

4.18.9 tz

The `tz` facet is used to annotate an `abstime`, `date`, or `time` object with a timezone. The value of a `tz` attribute is a *zoneinfo* string identifier. Zoneinfo is a standardized database sometimes referred to as the tz database or the Olsen database. It defines a set of time zone identifiers using the convention "*continent/city*". For example "America/New_York" identifies the time zone rules used by the east coast of the United Stated. UTC is represented as "Etc/UTC".

The zoneinfo database defines the current and historical rules for each zone including its offset from UTC and the rules for calculating daylight saving time. oBIX does not define a contract for modeling timezones, instead it just references the zoneinfo database using standard identifiers. It is up to oBIX enabled software to map zoneinfo identifiers to the UTC offset and daylight saving time rules.

The following rules are used to compute the timezone of an `abstime`, `date`, or `time` object:

1. If the `tz` attribute is specified, use it;
2. If the contract defines an inherited `tz` attribute, use it;
3. Assume the server's timezone as defined by the lobby's `About.tz`.

When using timezones, it is still required to specify the timezone offset within the value representation of an `abstime` or `time` object. It is an error condition for the `tz` facet to conflict with the timezone offset. For example New York has a -5 hour offset from UTC during standard time and a -4 hour offset during daylight saving time:

```
<abstime val="2007-12-25T12:00:00-05:00" tz="America/New_York"/>
<abstime val="2007-07-04T12:00:00-04:00" tz="America/New_York"/>
```

4.18.10 unit

The `unit` facet defines a unit of measurement. A unit attribute is a URI reference to a `obix:Unit` object (see section 12.5 for the contract definition). It is used with the `int` and `real` object types:

```
<real unit="obix:units/fahrenheit" val="67.2"/>
```

It is recommended that the `unit` facet not be overridden if declared in a contract. If it is overridden, then the override SHOULD use a `Unit` object with the same dimensions as the contract (it must measure the same physical quantity).

4.18.11 writable

The `writable` facet specifies if this object can be written by the client. If false (the default), then the object is read-only. It is used with all objects except operations and feeds:

```
<str name="userName" val="jsmith"     writable="false"/>
```

```
<str name="fullName" val="John Smith" writable="true"/>
```

5 Naming

All oBIX objects have two potential identifiers: name and href. Name is used to define the role of an object within its parent. Names are programmatic identifiers only; the `displayName` facet SHOULD be used for human interaction. Naming convention is to use camel case with the first character in lowercase. The primary purpose of names is to attach semantics to sub-objects. Names are also used to indicate overrides from a contract. A good analogy to names is the field/method names of a class in Java or C#.

Hrefs are used to attach URIs to objects. An href is always a *URI reference*, which means it might be a relative URI that requires normalization against a base URI. The exception to this rule is the href of the root object in an oBIX document – this href MUST be an absolute URI, not a URI reference. This allows the root object's href to be used as the effective base URI (xml:base) for normalization. A good analogy is hrefs in HTML or XLink.

Some objects may have both a name and an href, just a name, just an href, or neither. It is common for objects within a list to not use names, since most lists are unnamed sequences of objects. The oBIX specification makes a clear distinction between names and hrefs - you MUST NOT assume any relationship between names and hrefs. From a practical perspective many vendors will likely build an href structure that mimics the name structure, but client software MUST never assume such a relationship.

5.1 Name

The name of an object is represented using the `name` attribute. Names are programmatic identifiers with restrictions on their valid character set. A name SHOULD contain only ASCII letters, digits, underbar, or dollar signs. A digit MUST NOT be used as the first character. Convention is to use camel case with the first character in lower case: "foo", "fooBar", "thisIsOneLongName". Within a given object, all of its direct children MUST have unique names. Objects which don't have a `name` attribute are called *unnamed objects*. The root object of an oBIX document SHOULD NOT specify a `name` attribute (but almost always has an absolute href URI).

5.2 Href

The href of an object is represented using the `href` attribute. If specified, the root object MUST have an absolute URI. All other hrefs within an oBIX document are treated as URI references which may be relative. Because the root href is always an absolute URI, it may be used as the base for normalizing relative URIs within the document. The formal rules for URI syntax and normalization are defined in RFC 3986. We consider a few common cases that serve as design patterns within oBIX in Section 5.3.

As a general rule every object accessible for a read MUST specify a URI. An oBIX document returned from a read request MUST specify a root URI. However, there are certain cases where the object is transient, such as a computed object from an operation invocation. In these cases there MAY not be a root URI, meaning there is no way to retrieve this particular object again. If no root URI is provided, then the server's authority URI is implied to be the base URI for resolving relative URI references.

5.3 HTTP Relative URIs

Vendors are free to use any URI scheme, although the recommendation is to use HTTP URIs since they have well defined normalization semantics. This section provides a summary of how HTTP URI normalization should work within oBIX client agents. The general rules are:

- If the URI starts with "*scheme*:" then it is an globally absolute URI
- If the URI starts with a single slash, then it is server absolute URI
- If the URI starts with a "#", then it is a fragment identifier (discussed in next section)
- If the URI starts with "../", then the path must backup from the base

Otherwise the URI is assumed to be a relative path from the base URI

Some examples:

```
http://server/a    +  http://overthere/x  →  http://overthere/x
http://server/a    +  /x/y/z              →  http://server/x/y/z
http://server/a/b  +  c                   →  http://server/a/c
http://server/a/b/ +  c                   →  http://server/a/b/c
http://server/a/b  +  c/d                 →  http://server/a/c/d
http://server/a/b/ +  c/d                 →  http://server/a/b/c/d
http://server/a/b  +  ../c                →  http://server/c
http://server/a/b/ +  ../c                →  http://server/a/c
```

Perhaps one of the trickiest issues is whether the base URI ends with slash. If the base URI doesn't end with a slash, then a relative URI is assumed to be relative to the base's parent (to match HTML). If the base URI does end in a slash, then relative URIs can just be appended to the base. In practice, systems organized into hierarchical URIs SHOULD always specify the base URI with a trailing slash. Retrieval with and without the trailing slash SHOULD be supported with the resulting document always adding the implicit trailing slash in the root object's `href`.

5.4 Fragment URIs

It is not uncommon to reference an object internal to an oBIX document. This is achieved using fragment URI references starting with the "#". Let's consider the example:

```
<obj href="http://server/whatever/">
  <enum name="switch1" range="#onOff" val="on"/>
  <enum name="switch2" range="#onOff" val="off"/>
  <list is="obix:Range" href="onOff">
    <obj name="on"/>
    <obj name="off"/>
  </list>
</obj>
```

In this example there are two objects with a `range` facet referencing a fragment URI. Any URI reference starting with "#" MUST be assumed to reference an object within the same oBIX document. Clients SHOULD NOT perform another URI retrieval to dereference the object. In this case the object being referenced is identified via the `href` attribute.

In the example above the object with an href of "onOff" is both the target of the fragment URI, but also has the absolute URI "http://server/whatever/onOff". But suppose we had an object that was the target of a fragment URI within the document, but could not be directly addressed using an absolute URI? In that case the href attribute SHOULD be a fragment identifier itself. When an href attribute starts with "#" that means the only place it can be used is within the document itself:

```
…
  <list is="obix:Range" href="#onOff">
…
```

6 Contracts

Contracts are a mechanism to harness the inherit patterns in modeling oBIX data sources. What is a contract? Well basically it is just a normal oBIX object. What makes a contract object special, is that other objects reference it as a “template object” using the `is` attribute.

So what does oBIX use contracts for? Contracts solve many problems in oBIX:

- **Semantics**: contracts are used to define “types” within oBIX. This lets us collectively agree on common object definitions to provide consistent semantics across vendor implementations. For example the `Alarm` contract ensures that client software can extract normalized alarm information from any vendor’s system using the exact same object structure.
- **Defaults**: contracts also provide a convenient mechanism to specify default values. Note that when serializing object trees to XML (especially over a network), we typically don’t allow defaults to be used in order to keep client processing simple.
- **Type Export**: it is likely that many vendors will have a system built using a statically typed language like Java or C#. Contracts provide a standard mechanism to export type information in a format that all oBIX clients can consume.

Why use contracts versus other approaches? There are certainly lots of ways to solve the above problems. The benefit of the contract design is its flexibility and simplicity. Conceptually contracts provide an elegant model for solving many different problems with one abstraction. From a specification perspective, we can define new abstractions using the oBIX XML syntax itself. And from an implementation perspective, contracts give us a machine readable format that clients already know how to retrieve and parse – to use OO lingo, the exact same syntax is used to represent both a class and an instance.

6.1 Contract Terminology

In order to discuss contracts, it is useful to define a couple of terms:

- **Contract**: is a reusable object definition expressed as a standard oBIX XML document. Contracts are the templates or prototypes used as the foundation of the oBIX type system.
- **Contract List**: is a list of one or more URIs to contract objects. It is used as the value of the `is`, `of`, `in` and `out` attributes. The list of URIs is separated by the space character. You can think of a contract list as a type declaration.
- **Implements**: when an object specifies a contract in its contract list, the object is said to *implement* the contract. This means that the object is inheriting both the structure and semantics of the specified contract.
- **Implementation**: an object which implements a contract is said to be an *implementation* of that contract.

6.2 Contract List

The syntax of a contract list attribute is a list of URI references to other oBIX objects. It is used as the value of the `is`, `of`, `in` and `out` attributes. The URIs within the list are separated by the space character (Unicode 0x20). Just like the href attribute, a contract URI can be an absolute URI, server relative, or even a fragment reference. The URIs within a contract list may be scoped with an XML namespace prefix (see Section 7.6).

6.3 Is Attribute

An object defines the contracts it implements via the is attribute. The value of the is attribute is a contract list. If the is attribute is unspecified, then the following rules are used to determine the implied contract list:

- If the object is an item inside a list or feed, then the contract list specified by the of attribute is used.
- If the object overrides (by name) an object specified in one of its contracts, then the contract list of the overridden object is used.
- If all the above rules fail, then the respective primitive contract is used. For example, an obj element has an implied contract of obix:obj and real an implied contract of obj:real.

Note that element names such as bool, int, or str are syntactic sugar for an implied contract. However if an object implements one of the primitives, then it MUST use the correct XML element name. For example if an object implements obix:int, then it must be expressed as <int/>, rather than <obj is="obix:int"/>. Therefore it is invalid to implement multiple value types - such as implementing both obix:bool and obix:int.

6.4 Contract Inheritance

Contracts are a mechanism of inheritance – they establish the classic “is a” relationship. In the abstract sense a contract allows us to inherit a *type*. We can further distinguish between the explicit and implicit contract:

- **Explicit Contract**: defines an object structure which all implementations must conform with.
- **Implicit Contract**: defines semantics associated with the contract. Usually the implicit contract is documented using natural language prose. It isn’t mathematical, but rather subject to human interpretation.

For example when we say an object implements the Alarm contract, we immediately know that will have a child called timestamp. This structure is in the explicit contract of Alarm and is formally defined in XML. But we also attach semantics to what it means to be an Alarm object: that the object is providing information about an alarm event. These fuzzy concepts can’t be captured in machine language; rather they can only be captured in prose.

When an object declares itself to implement a contract it must meet both the explicit contract and the implicit contract. An object shouldn’t put obix:Alarm in its contract list unless it really represents an alarm event. There isn’t much more to say about implicit contracts other than it is recommended that a human brain be involved. So now let’s look at the rules governing the explicit contract.

A contract’s named children objects are automatically applied to implementations. An implementation may choose to *override* or *default* each of its contract’s children. If the implementation omits the child, then it is assumed to default to the contract’s value. If the implementation declares the child (by name), then it is overridden and the implementation’s value should be used. Let’s look at an example:

```
<obj href="/def/television">
  <bool name="power"   val="false"/>
  <int  name="channel" val="2" min="2" max="200"/>
</obj>

<obj href="/livingRoom/tv" is="/def/television">
  <int name="channel" val="8"/>
  <int name="volume"  val="22"/>
</obj>
```

In this example we have a contract object identified with the URI “/def/television”. It has two children to store power and channel. Then we specify a living room TV instance that includes “/def/television” in its contract list via the is attribute. In this object, channel is *overridden* to 8 from its default value of 2. However since power was omitted, it is implied to *default* to false.

An override is always matched to its contract via the `name` attribute. In the example above we knew we were overriding channel, because we declared an object with a name of "channel". We also declared an object with a name of "volume". Since volume wasn't declared in the contract, we assume it's a new definition specific to this object.

Also note that the contract's channel object declares a `min` and `max` facet. These two facets are also inherited by the implementation. Almost all attributes are inherited from their contract including facets, `val`, `of`, `in`, and `out`. The `href` attribute are never inherited. The `null` attribute inherits as follows:

1. If the `null` attribute is specified, then its explicit value is used;
2. If a `val` attribute is specified and `null` is unspecified, then `null` is implied to be false;
3. If neither a `val` attribute or a `null` attribute is specified, then the `null` attribute is inherited from the contract;

This allows us to implicitly override a null object to non-null without specifying the `null` attribute.

6.5 Override Rules

Contract overrides are required to obey the implicit and explicit contract. Implicit means that the implementation object provides the same semantics as the contract it implements. In the example above it would be incorrect to override channel to store picture brightness. That would break the semantic contract.

Overriding the explicit contract means to override the value, facets, or contract list. However we can never override the object to be in incompatible value type. For example if the contract specifies a child as `real`, then all implementations must use `real` for that child. As a special case, `obj` may be narrowed to any other element type.

We also have to be careful when overriding attributes to never break restrictions the contract has defined. Technically this means we can *specialize* or *narrow* the value space of a contract, but never *generalize* or *widen* it. This concept is called *covariance*. Let's take our example from above:

```
<int name="channel" val="2" min="2" max="200"/>
```

In this example the contract has declared a value space of 2 to 200. Any implementation of this contract must meet this restriction. For example it would an error to override `min` to –100 since that would widen the value space. However we can narrow the value space by overriding `min` to a number greater than 2 or by overriding `max` to a number less than 200. The specific override rules applicable to each facet are documented in section 4.18.

6.6 Multiple Inheritance

An object's contract list may specify multiple contract URIs to implement. This is actually quite common - even required in many cases. There are two topics associated with the implementation of multiple contracts:

- **Flattening**: contract lists SHOULD always be *flattened* when specified. This comes into play when a contract has its own contract list (Section 6.6.1).
- **Mixins**: the mixin design specifies the exact rules for how multiple contracts are merged together. This section also specifies how conflicts are handled when multiple contracts contain children with the same name (Section 6.6.2).

6.6.1 Flattening

It is common for contract objects themselves to implement contracts, just like it is common in OO languages to chain the inheritance hierarchy. However due to the nature of accessing oBIX documents over a network, we wish to minimize round trip network requests which might be required to "learn" about a complex contract hierarchy. Consider this example:

```
<obj href="/A" />
<obj href="/B" is="/A" />
<obj href="/C" is="/B" />
```

```
<obj href="/D" is="/C" />
```

In this example if we were reading object D for the first time, it would take three more requests to fully learn what contracts are implemented (one for C, B, and A). Furthermore, if our client was just looking for objects that implemented B, it would difficult to determine this just by looking at D.

Because of these issues, servers are required to flatten their contract inheritance hierarchy into a list when specifying the `is`, `of`, `in`, or `out` attributes. In the example above, the correct representation would be:

```
<obj href="/A" />
<obj href="/B" is="/A" />
<obj href="/C" is="/B /A" />
<obj href="/D" is="/C /B /A" />
```

This allows clients to quickly scan Ds contract list to see that D implements C, B, and A without further requests.

6.6.2 Mixins

Flattening is not the only reason a contract list might contain multiple contract URIs. oBIX also supports the more traditional notion of multiple inheritance using a mixin metaphor. Consider the following example:

```
<obj href="acme:Device">
  <str name="serialNo"/>
</obj>

<obj href="acme:Clock" is="acme:Device">
  <op name="snooze">
  <int name="volume" val="0"/>
</obj>

<obj href="acme:Radio" is="acme:Device ">
  <real name="station" min="87.0" max="107.5">
  <int name="volume" val="5"/>
</obj>

<obj href="acme:ClockRadio" is="acme:Radio acme:Clock acme:Device"/>
```

In this example `ClockRadio` implements both `Clock` and `Radio`. Via flattening of `Clock` and `Radio`, `ClockRadio` also implements `Device`. In oBIX this is called a *mixin* – `Clock`, `Radio`, and `Device` are mixed into (merged into) `ClockRadio`. Therefore `ClockRadio` inherits four children: `serialNo`, `snooze`, `volume`, and `station`. Mixins are a form of multiple inheritance akin to Java/C# interfaces (remember oBIX is about the type inheritance, not implementation inheritance).

Note that `Clock` and `Radio` both implement `Device` - the classic diamond inheritance pattern. From `Device`, `ClockRadio` inherits a child named `serialNo`. Furthermore notice that both `Clock` and `Radio` declare a child named `volume`. This naming collision could potentially create confusion for what `serialNo` and `volume` mean in `ClockRadio`.

In oBIX we solve this problem by flattening the contract's children using the following rules:

1. Process the contract definitions in the order they are listed
2. If a new child is discovered, it is mixed into the object's definition
3. If a child is discovered we already processed via a previous contract definition, then the previous definition takes precedence. However it is an error if the duplicate child is not *contract compatible* with the previous definition (see Section 6.7).

In the example above this means that `Radio.volume` is the definition we use for `ClockRadio.volume`, because `Radio` has a higher precedence than `Clock` (it is first in the contract list). Thus `ClockRadio.volume` has a default value of "5". However it would be invalid if `Clock.volume` were declared as `str`, since it would not be contract compatible with `Radio`'s definition as an `int` – in that case `ClockRadio` could not implement both `Clock` and `Radio`. It is the server vendor's responsibility not to create incompatible name collisions in contracts.

The first contract in a list is given specific significance since its definition trumps all others. In oBIX this contract is called the *primary contract*. It is recommended that the primary contract implement all the other contracts specified in the contract list (this actually happens quite naturally by itself in many programming languages). This makes it easier for clients to bind the object into a strongly typed class if desired. Contracts MUST NOT implement themselves nor have circular inheritance dependencies.

6.7 Contract Compatibility

A contract list which is covariantly substitutable with another contract list is said to be *contract compatible*. Contract compatibility is a useful term when talking about mixin rules and overrides for lists and operations. It is a fairly common sense notion similar to previously defined override rules – however, instead of the rules applied to individual facet attributes, we apply it to an entire contract list.

A contract list X is compatible with contract list Y, if and only if X narrows the value space defined by Y. This means that X can narrow the set of objects which implement Y, but never expand the set. Contract compatibility is not commutative (X is compatible with Y does not imply Y is compatible with X). If that definition sounds too highfalutin, you can boil it down to this practical rule: X can add new URIs to Y's list, but never take any away.

6.8 Lists (and Feeds)

Implementations derived from `list` or `feed` contracts inherit the `of` attribute. Like other attributes we can override the `of` attribute, but only if contract compatible - you SHOULD include all of the URIs in the contract's `of` attribute, but you can add additional ones (see Section 6.7).

Lists and feeds also have the special ability to implicitly define the contract list of their contents. In the following example it is implied that each child element has a contract list of `/def/MissingPerson` without actually specifying the `is` attribute in each list item:

```
<list of="/def/MissingPerson">
  <obj> <str name="fullName" val="Jack Shephard"/> </obj>
  <obj> <str name="fullName" val="John Locke"/> </obj>
  <obj> <str name="fullName" val="Kate Austen"/> </obj>
</list>
```

If an element in the list or feed does specify its own `is` attribute, then it MUST be contract compatible with the `of` attribute.

If you wish to specify that a list should contain references to a given type, then you SHOULD include `obix:ref` in the `of` attribute. For example, to specify that a list should contain references to obix:History objects (as opposed to inline History objects):

```
<list name="histories" of="obix:ref obix:History"/>
```

7 XML Encoding

Chapter 4 specifies an abstract object model used to standardize how oBIX information is modeled. This chapter specifies how the object model is encoded in XML.

7.1 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.

7.2 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 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.

7.3 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 7.6).

7.4 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).

7.5 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.

7.6 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"/>
```

8 Binary Encoding

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.

8.1 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.

8.2 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

8.3 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).

8.3.1 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.

8.3.2 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.

8.3.3 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
```

8.3.4 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 8.5). 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.

8.3.5 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.

8.3.6 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
```

8.3.7 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
```

8.3.8 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`

8.3.9 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.

8.4 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.

8.4.1 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 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
```

8.5 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
```

9 Operations

Operations are the things that you can "do" to an oBIX object. They are akin to methods in traditional OO languages. Typically they map to commands rather than a variable that has continuous state. Unlike value objects which represent an object and its current state, the `op` element merely represents the definition of an operation you can invoke.

All operations take exactly one object as a parameter and return exactly one object as a result. The `in` and `out` attributes define the contract list for the input and output objects. If you need multiple input or output parameters, then wrap them in a single object using a contract as the signature. For example:

```
<op href="/addTwoReals" in="/def/AddIn" out="obix:real"/>

<obj href="/def/AddIn">
  <real name="a"/>
  <real name="b"/>
</obj>
```

Objects can override the operation definition from one of their contracts. However the new `in` or `out` contract list MUST be contract compatible (see Section 6.7) with the contract's definition.

If an operation doesn't require a parameter, then specify `in` as `obix:Nil`. If an operation doesn't return anything, then specify `out` as `obix:Nil`. Occasionally an operation is inherited from a contract which is unsupported in the implementation. In this case set the `status` attribute to `disabled`.

Operations are always invoked via their own `href` attribute (not their parent's `href`). Therefore operations SHOULD always specify an `href` attribute if you wish clients to invoke them. A common exception to this rule is contract definitions themselves.

10 Object Composition

A good metaphor for comparison with oBIX is the World Wide Web. If you ignore all the fancy stuff like JavaScript and Flash, basically the WWW is a web of HTML documents hyperlinked together with URIs. If you dive down one more level, you could say the WWW is a web of HTML elements such as `<p>`, `<table>`, and `<div>`.

What the WWW does for HTML documents, oBIX does for objects. The logical model for oBIX is a global web of oBIX objects linked together via URIs. Some of these oBIX objects are static documents like contracts or device descriptions. Other oBIX objects expose real-time data or services. But they all are linked together via URIs to create the *oBIX Web*.

Individual objects are composed together in two ways to define this web. Objects may be composed together via *containment* or via *reference*.

10.1 Containment

Any oBIX object may contain zero or more children objects. This even includes objects which might be considered primitives such as `bool` or `int`. All objects are open ended and free to specify new objects which may not be in the object's contract. Containment is represented in the XML syntax by nesting the XML elements:

```
<obj href="/a/">
  <list name="b" href="b">
    <obj href="b/c">
  </list>
</obj>
```

In this example the object identified by "/a" contains "/a/b", which in turn contains "/a/b/c". Child objects may be named or unnamed depending on if the `name` attribute is specified (Section 5.1). In the example, "/a/b" is named and "/a/b/c" is unnamed. Typically named children are used to represent fields in a record, structure, or class type. Unnamed children are often used in lists.

10.2 References

Let's go back to our WWW metaphor. Although the WWW is a web of individual HTML elements like `<p>` and `<div>`, we don't actually pass individual <p> elements around over the network. Rather we "chunk" them into HTML documents and always pass the entire document over the network. To tie it all together, we create links between documents using the `<a>` anchor element. These anchors serve as place holders, referencing outside documents via a URI.

A oBIX reference is basically just like an HTML anchor. It serves as placeholder to "link" to another oBIX object via a URI. While containment is best used to model small trees of data, references may be used to model very large trees or graphs of objects. As a matter fact, with references we can link together all oBIX objects on the Internet to create the oBIX Web.

10.3 Extents

When oBIX is applied to a problem domain, we have to decide whether to model relationships using either containment or references. These decisions have a direct impact on how your model is represented in XML and accessed over the network. The containment relationship is imbued with special semantics regarding XML encoding and eventing. In fact, oBIX coins a term for containment called an object's *extent*. An object's extent is its tree of children down to references. Only objects which have an href have an extent. Objects without an href are always included in one or more of their ancestors extents.

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

```
  <ref name="e" href="/e"/>
</obj>
```

In the example above, we have five objects named ‘a’ to ‘e’. Because ‘a’ includes an href, it has an associated extent, which encompasses ‘b’ and ‘c’ by containment and ‘d’ and ‘e’ by reference. Likewise, ‘b’ has an href which results in an extent encompassing ‘c’ by containment and ‘d’ by reference. Object ‘c’ does not provide a direct href, but exists in both the ‘a’ and ‘b’ objects’ extents. Note an object with an href has exactly one extent, but can be nested inside multiple extents.

10.4 XML

When marshaling objects into an XML, it is required that an extent always be fully inlined into the XML document. The only valid objects which may be referenced outside the document are `ref` element themselves.

If the object implements a contract, then it is required that the extent defined by the contract be fully inlined into the document (unless the contract itself defined a child as a `ref` element). An example of a contract which specifies a child as a `ref` is `Lobby.about` (11.3).

11 Networking

The heart of oBIX is its object model and associated XML syntax. However, the primary use case for oBIX is to access information and services over a network. The oBIX architecture is based on a client/server network model:

- **Server**: software containing oBIX enabled data and services. Servers respond to requests from client over a network.
- **Client**: software which makes requests to servers over a network to access oBIX enabled data and services.

There is nothing to prevent software from being both an oBIX client and server. Although a key tenant of oBIX is that a client is not required to implement server functionality which might require a server socket to accept incoming requests.

11.1 Request / Response

All network access is boiled down into three request / response types:

- **Read**: return the current state of an object at a given URI as an oBIX XML document.
- **Write**: update the state of an existing object at a URI. The state to write is passed over the network as an oBIX XML document. The new updated state is returned in an oBIX XML document.
- **Invoke**: invoke an operation identified by a given URI. The input parameter and output result are passed over the network as an oBIX XML document.

Exactly how these three request/responses are implemented between a client and server is called a *protocol binding*. The oBIX specification defines two standard protocol bindings: HTTP Binding (see Chapter 18) and SOAP Binding (see Chapter 19). However all protocol bindings must follow the same read, write, invoke semantics discussed next.

11.1.1 Read

The read request specifies an object's URI and the read response returns the current state of the object as an oBIX document. The response MUST include the object's complete extent (see 10.3). Servers may return an `err` object to indicate the read was unsuccessful – the most common error is `obix:BadUriErr` (see 11.2 for standard error contracts).

11.1.2 Write

The write request is designed to overwrite the current state of an existing object. The write request specifies the URI of an existing object and its new desired state. The response returns the updated state of the object. If the write is successful, the response MUST include the object's complete extent (see 10.3). If the write is unsuccessful, then the server MUST return an `err` object indicating the failure.

The server is free to completely or partially ignore the write, so clients SHOULD be prepared to examine the response to check if the write was successful. Servers may also return an `err` object to indicate the write was unsuccessful.

Clients are not required to include the object's full extent in the request. Objects explicitly specified in the request object tree SHOULD be overwritten or "overlaid" over the server's actual object tree. Only the `val` attribute should be specified for a write request (outside of identification attributes such as `name`). The `null` attribute MAY also be used to set an object to null. If the `null` attribute is not specified and the `val` attribute is specified, then it is implied that null is false. A write operation that provides facets has unspecified behavior. When writing `int` or `reals` with `units`, the write value MUST be in the same units as the server specifies in read requests – clients MUST NOT provide a different `unit` facet and expect the server to auto-convert (in fact the `unit` facet SHOULD NOT be included in the request).

11.1.3 Invoke

The invoke request is designed to trigger an operation. The invoke request specified the URI of an `op` object and the input argument object. The response includes the output object. The response MUST include the output object's complete extent (see 10.3). Servers MAY instead return an `err` object to indicate the invoke was unsuccessful.

11.2 Errors

Request errors are conveyed to clients with the `err` element. Any time an oBIX server successfully receives a request and the request cannot be processed, then the server SHOULD return an `err` object to the client. Returning a valid oBIX document with `err` SHOULD be used when feasible rather than protocol specific error handling (such as an HTTP response code). Such a design allows for consistency with batch request partial failures and makes protocol binding more pluggable by separating data transport from application level error handling.

A few contracts are predefined for common errors:

- **BadUriErr**: used to indicate either a malformed URI or a unknown URI;
- **UnsupportedErr**: used to indicate an a request which isn't supported by the server implementation (such as an operation defined in a contract, which the server doesn't support);
- **PermissionErr**: used to indicate that the client lacks the necessary security permission to access the object or operation.

The contracts for these errors are:

```
<err href="obix:BadUriErr"/>
<err href="obix:UnsupportedErr"/>
<err href="obix:PermissionErr"/>
```

If one of the above contracts makes sense for an error, then it SHOULD be included in the `err` element's `is` attribute. It is strongly encouraged to also include a useful description of the problem in the `display` attribute.

11.3 Lobby

All oBIX servers MUST provide an object which implements `obix:Lobby`. The `Lobby` object serves as the central entry point into an oBIX server, and lists the URIs for other well-known objects defined by the oBIX specification. Theoretically all a client needs to know to bootstrap discovery is one URI for the `Lobby` instance. By convention this URI is "http://server/obix", although vendors are certainly free to pick another URI. The Lobby contract is:

```
<obj href="obix:Lobby">
  <ref name="about" is="obix:About"/>
  <op  name="batch" in="obix:BatchIn" out="obix:BatchOut"/>
  <ref name="watchService" is="obix:WatchService"/>
</obj>
```

The `Lobby` instance is where vendors SHOULD place vendor specific objects used for data and service discovery.

11.4 About

The `obix:About` object is a standardized list of summary information about an oBIX server. Clients can discover the `About` URI directly from the `Lobby`. The `About` contract is:

```
<obj href="obix:About">

  <str name="obixVersion"/>

  <str name="serverName"/>
  <abstime name="serverTime"/>
  <abstime name="serverBootTime"/>

```

```
  <str name="vendorName"/>
  <uri name="vendorUrl"/>

  <str name="productName"/>
  <str name="productVersion"/>
  <uri name="productUrl"/>

  <str name="tz"/>
</obj>
```

The following children provide information about the oBIX implementation:

- **obixVersion**: specifies which version of the oBIX specification the server implements. This string MUST be a list of decimal numbers separated by the dot character (Unicode 0x2E). The current version string is “1.1”.

The following children provide information about the server itself:

- **serverName**: provides a short localized name for the server.
- **serverTime**: provides the server’s current local time.
- **serverBootTime**: provides the server’s start time - this SHOULD be the start time of the oBIX server software, not the machine’s boot time.

The following children provide information about the server’s software vendor:

- **vendorName**: the company name of the vendor who implemented the oBIX server software.
- **vendorUrl**: a URI to the vendor’s website.

The following children provide information about the software product running the server:

- **productName**: with the product name of oBIX server software.
- **productUrl**: a URI to the product’s website.
- **productVersion**: a string with the product’s version number. Convention is to use decimal digits separated by dots.

The following children provide additional miscellaneous information:

- **tz**: specifies a zoneinfo identifier for the server’s default timezone.

11.5 Batch

The `Lobby` defines a `batch` operation which is used to batch multiple network requests together into a single operation. Batching multiple requests together can often provide significant performance improvements over individual round-robin network requests. As a general rule, one big request will always out-perform many small requests over a network.

A batch request is an aggregation of read, write, and invoke requests implemented as a standard oBIX operation. At the protocol binding layer, it is represented as a single invoke request using the `Lobby.batch` URI. Batching a set of requests to a server MUST be processed semantically equivalent to invoking each of the requests individually in a linear sequence.

The batch operation inputs a `BatchIn` object and outputs a `BatchOut` object:

```
<list href="obix:BatchIn" of="obix:uri"/>

<list href="obix:BatchOut" of="obix:obj"/>
```

The `BatchIn` contract specifies a list of requests to process identified using the `Read`, `Write`, or `Invoke` contract:

```
<uri href="obix:Read"/>

<uri href="obix:Write">
  <obj name="in"/>
</uri>
```

```
<uri href="obix:Invoke">
  <obj name="in"/>
</uri>
```

The `BatchOut` contract specifies an ordered list of the response objects to each respective request. For example the first object in `BatchOut` must be the result of the first request in `BatchIn`. Failures are represented using the `err` object. Every `uri` passed via `BatchIn` for a read or write request MUST have a corresponding result `obj` in `BatchOut` with an `href` attribute using an identical string representation from `BatchIn` (no normalization or case conversion is allowed).

It is up to vendors to decide how to deal with partial failures. In general idempotent requests SHOULD indicate a partial failure using `err`, and continue processing additional requests in the batch. If a server decides not to process additional requests when an error is encountered, then it is still REQUIRED to return an `err` for each respective request not processed.

Let's look at a simple example:

```
<list is="obix:BatchIn">
  <uri is="obix:Read" val="/someStr"/>
  <uri is="obix:Read" val="/invalidUri"/>
  <uri is="obix:Write" val="/someStr">
    <str name="in" val="new string value"/>
  </uri>
</list>

<list is="obix:BatchOut">
  <str href="/someStr" val="old string value"/>
  <err href="/invalidUri" is="obix:BadUriErr" display="href not found"/>
  <str href="/someStr" val="new string value">
</list>
```

In this example, the batch request is specifying a read request for "/someStr" and "/invalidUri", followed by a write request to "/someStr". Note that the write request includes the value to write as a child named "in".

The server responds to the batch request by specifying exactly one object for each request URI. The first read request returns a `str` object indicating the current value identified by "/someStr". The second read request contains an invalid URI, so the server returns an `err` object indicating a partial failure and continues to process subsequent requests. The third request is a write to "someStr". The server updates the value at "someStr", and returns the new value. Note that because the requests are processed in order, the first request provides the original value of "someStr" and the third request contains the new value. This is exactly what we would expect had we processed each of these requests individually.

12 Core Contract Library

This chapter defines some fundamental object contracts that serve as building blocks for the oBIX specification.

12.1 Nil

The `obix:Nil` contract defines a standardized null object. Nil is commonly used for an operation's `in` or `out` attribute to denote the absence of an input or output. The definition:

```
<obj href="obix:Nil" null="true"/>
```

12.2 Range

The `obix:Range` contract is used to define a `bool` or `enum`'s range. `Range` is a list object that contains zero or more objects called the range items. Each item's `name` attribute specifies the identifier used as the literal value of an `enum`. Item ids are never localized, and MUST be used only once in a given range. You may use the optional `displayName` attribute to specify a localized string to use in a user interface. The definition of `Range`:

```
<list href="obix:Range" of="obix:obj"/>
```

An example:

```
<list href="/enums/OffSlowFast" is="obix:Range">
  <obj name="off"  displayName="Off"/>
  <obj name="slow" displayName="Slow Speed"/>
  <obj name="fast" displayName="Fast Speed"/>
</list>
```

The range facet may be used to define the localized text of a `bool` value using the ids of "true" and "false":

```
<list href="/enums/OnOff" is="obix:Range">
  <obj name="true"  displayName="On"/>
  <obj name="false" displayName="Off"/>
</list >
```

12.3 Weekday

The `obix:Weekday` contract is a standardized enum for the days of the week:

```
<enum href="obix:Weekday" range="#Range">
  <list href="#Range" is="obix:Range">
    <obj name="sunday" />
    <obj name="monday" />
    <obj name="tuesday" />
    <obj name="wednesday" />
    <obj name="thursday" />
    <obj name="friday" />
    <obj name="saturday" />
  </list>
</enum>
```

12.4 Month

The `obix:Month` contract is a standardized enum for the months of the year:

```
<enum href="obix:Month" range="#Range">
  <list href="#Range" is="obix:Range">
    <obj name="january" />
    <obj name="febuary" />
    <obj name="march" />
    <obj name="april" />
    <obj name="may" />
```

```
    <obj name="june" />
    <obj name="july" />
    <obj name="august" />
    <obj name="september" />
    <obj name="october"  />
    <obj name="november" />
    <obj name="december" />
  </list>
</enum>
```

12.5 Units

Representing units of measurement in software is a thorny issue. oBIX provides a unit framework for mathematically defining units within the object model. An extensive database of predefined units is also provided.

All units measure a specific quantity or dimension in the physical world. Most known dimensions can be expressed as a ratio of the seven fundamental dimensions: length, mass, time, temperature, electrical current, amount of substance, and luminous intensity. These seven dimensions are represented in SI respectively as kilogram (kg), meter (m), second (sec), Kelvin (K), ampere (A), mole (mol), and candela (cd).

The `obix:Dimension` contract defines the ratio of the seven SI units using a positive or negative exponent:

```
<obj href="obix:Dimension">
  <int name="kg"  val="0"/>
  <int name="m"   val="0"/>
  <int name="sec" val="0"/>
  <int name="K"   val="0"/>
  <int name="A"   val="0"/>
  <int name="mol" val="0"/>
  <int name="cd"  val="0"/>
</obj>
```

A `Dimension` object contains zero or more ratios of `kg`, `m`, `sec`, `K`, `A`, `mol`, or `cd`. Each of these ratio maps to the exponent of that base SI unit. If a ratio is missing then the default value of zero is implied. For example acceleration is m/s^2, which would be encoded in oBIX as:

```
<obj is="obix:Dimension">
  <int name="m"   val="1"/>
  <int name="sec" val="-2"/>
</obj>
```

Units with equal dimensions are considered to measure the same physical quantity. This is not always precisely true, but is good enough for practice. This means that units with the same dimension are convertible. Conversion can be expressed by specifying the formula required to convert the unit to the dimension’s normalized unit. The normalized unit for every dimension is the ratio of SI units itself. For example the normalized unit of energy is the joule $m^2 \bullet kg \bullet s^{-2}$. The kilojoule is 1000 joules and the watt-hour is 3600 joules. Most units can be mathematically converted to their normalized unit and to other units using the linear equations:

```
unit = dimension • scale + offset
toNormal = scalar • scale + offset
fromNormal = (scalar - offset) / scale
toUnit = fromUnit.fromNormal( toUnit.toNormal(scalar) )
```

There are some units which don’t fit this model including logarithm units and units dealing with angles. But this model provides a practical solution for most problem spaces. Units which don’t fit this model SHOULD use a dimension where every exponent is set to zero. Applications SHOULD NOT attempt conversions on these types of units.

The `obix:Unit` contract defines a unit including its dimension and its toNormal equation:

```
<obj href="obix:Unit">
  <str  name="symbol"/>
  <obj  name="dimension" is="obix:Dimension"/>
  <real name="scale" val="1"/>
```

```
  <real name="offset" val="0"/>
</obj>
```

The `unit` element contains a `symbol`, `dimension`, `scale`, and `offset` sub-object:

- **`symbol`**: The `symbol` element defines a short abbreviation to use for the unit. For example "°F" would be the symbol for degrees Fahrenheit. The `symbol` element SHOULD always be specified.
- **`dimension`**: The `dimension` object defines the dimension of measurement as a ratio of the seven base SI units. If omitted, the `dimension` object defaults to the `obix:Dimension` contract, in which case the ratio is the zero exponent for all seven base units.
- **`scale`**: The `scale` element defines the scale variable of the toNormal equation. The `scale` object defaults to 1.
- **`offset`**: The `offset` element defines the offset variable of the toNormal equation. If omitted then `offset` defaults to 0.

The `display` attribute SHOULD be used to provide a localized full name for the unit based on the client's locale. If the `display` attribute is omitted, clients SHOULD use `symbol` for display purposes.

An example for the predefined unit for kilowatt:

```
• <obj href="obix:units/kilowatt" display="kilowatt">
  <str name="symbol" val="kW"/>
  <obj name="dimension">
    <int name="m" val="2"/>
    <int name="kg" val="1"/>
    <int name="sec" val="-3"/>
  </obj>
  <real name="scale" val="1000"/>
</obj>
```

Automatic conversion of units is considered a localization issue – see Section 18.5 for more details.

13 Watches

A key requirement of oBIX is access to real-time information. We wish to enable clients to efficiently receive access to rapidly changing data. However, we don't want to require clients to implement web servers or expose a well-known IP address. In order to address this problem, oBIX provides a model for client polled eventing called *watches*. The watch lifecycle is as follows:

- The client creates a new watch object with the `make` operation on the server's `WatchService` URI. The server defines a new `Watch` object and provides a URI to access the new watch.
- The client registers (and unregisters) objects to watch using operations on the `Watch` object.
- The client periodically polls the `Watch` URI using the `pollChanges` operation to obtain the events which have occurred since the last poll.
- The server frees the `Watch` under two conditions. The client may explicitly free the `Watch` using the `delete` operation. Or the server may automatically free the `Watch` because the client fails to poll after a predetermined amount of time (called the lease time).

Watches allow a client to maintain a real-time cache for the current state of one or more objects. They are also used to access an event stream from a `feed` object. Plus, watches serve as the standardized mechanism for managing per-client state on the server via leases.

13.1 WatchService

The `WatchService` object provides a well-known URI as the factory for creating new watches. The `WatchService` URI is available directly from the `Lobby` object. The contract for `WatchService`:

```
<obj href="obix:WatchService">
  <op name="make" in="obix:Nil" out="obix:Watch"/>
</obj>
```

The make operation returns a new empty Watch object as an output. The href of the newly created Watch object can then be used for invoking operations to populate and poll the data set.

13.2 Watch

`Watch` object is used to manage a set of objects which are subscribed and periodically polled by clients to receive the latest events. The contract is:

```
<obj href="obix:Watch">
  <reltime name="lease" min="PT0S" writable="true"/>
  <op name="add"    in="obix:WatchIn" out="obix:WatchOut"/>
  <op name="remove" in="obix:WatchIn"/>
  <op name="pollChanges" out="obix:WatchOut"/>
  <op name="pollRefresh" out="obix:WatchOut"/>
  <op name="delete"/>
</obj>

<obj href="obix:WatchIn">
  <list name="hrefs" of="obix:WatchInItem"/>
</obj>

<uri href="obix:WatchInItem">
  <obj name="in"/>
</uri>

<obj href="obix:WatchOut">
  <list name="values" of="obix:obj"/>
</obj>
```

Many of the Watch operations use two contracts: `obix:WatchIn` and `obix:WatchOut`. The client identifies objects to `add` and `remove` from the poll list via WatchIn. This object contains a list of URIs. Typically these URIs SHOULD be server relative.

The server responds to `add`, `pollChanges`, and `pollRefresh` operations via the WatchOut contract. This object contains the list of subscribed objects - each object MUST specify an href URI using the exact same string as the URI identified by the client in the corresponding WatchIn. Servers are not allowed to perform any case conversions or normalization on the URI passed by the client. This allows client software to use the URI string as a hash key to match up server responses.

13.2.1 Watch.add

Once a Watch has been created, the client can add new objects to watch using the `add` operation. This operation inputs a list of URIs and outputs the current value of the objects referenced. The objects returned are required to specify an href using the exact string representation input by the client. If any object cannot be processed, then a partial failure SHOULD be expressed by returning an `err` object with the respective href. Subsequent URIs MUST NOT be effected by the failure of one invalid URI. The `add` operation MUST never return objects not explicitly included in the input URIs (even if there are already existing objects in the watch list). No guarantee is made that the order of objects in `WatchOut` match the order in of URIs in `WatchIn` – clients must use the URI as a key for matching.

Note that the URIs supplied via WatchIn may include an optional `in` parameter. This parameter is only used when subscribing a watch to a `feed` object. Feeds also differ from other objects in that they return a list of historic events in WatchOut. Feeds are discussed in detail in Section 13.4.

It is invalid to add an `op`’s href to a watch, the server MUST report an err.

If an attempt is made to add a URI to a watch which was previously already added, then the server SHOULD return the current object’s value in the `WatchOut` result, but treat poll operations as if the URI was only added once – polls SHOULD only return the object once. If an attempt is made to add the same URI multiple times in the same `WatchIn` request, then the server SHOULD only return the object once.

Note: the lack of a trailing slash can cause problems with watches. Consider a client which adds a URI to a watch without a trailing slash. The client will use this URI as a key in its local hashtable for the watch. Therefore the server MUST use the URI exactly as the client specified. However, if the object’s extent includes children objects they will not be able to use relative URIs. It is RECOMMENDED that servers fail-fast in these cases and return a BadUriErr when clients attempt to add a URI without a trailing slash to a watch (even though they may allow it for a normal read request).

13.2.2 Watch.remove

The client can remove objects from the watch list using the `remove` operation. A list of URIs is input to `remove`, and the Nil object is returned. Subsequent `pollChanges` and `pollRefresh` operations MUST cease to include the specified URIs. It is possible to remove every URI in the watch list; but this scenario MUST NOT automatically free the Watch, rather normal poll and lease rules still apply. It is invalid to use the `WatchInItem.in` parameter for a `remove` operation.

13.2.3 Watch.pollChanges

Clients SHOULD periodically poll the server using the `pollChanges` operation. This operation returns a list of the subscribed objects which have changed. Servers SHOULD only return the objects which have been modified since the last poll request for the specific Watch. As with `add`, every object MUST specify an href using the exact same string representation the client passed in the original `add` operation. The entire extent of the object SHOULD be returned to the client if any one thing inside the extent has changed on the server side.

Invalid URIs MUST never be included in the response (only in `add` and `pollRefresh`). An exception to this rule is when an object which is valid is removed from the URI space. Servers SHOULD indicate an object has been removed via an `err` with the `BadUriErr` contract.

13.2.4 Watch.pollRefresh

The `pollRefresh` operation forces an update of every object in the watch list. The server MUST return every object and it's full extent in the response using the href with the exact same string representation passed by the client in the original `add`. Invalid URIs in the poll list SHOULD be included in the response as an `err` element. A `pollRefresh` resets the poll state of every object, so that the next `pollChanges` only returns objects which have changed state since the `pollRefresh` invocation.

13.2.5 Watch.lease

All Watches have a *lease time*, specified by the `lease` child. If the lease time elapses without the client initiating a request on the Watch, then the server is free to *expire* the watch. Every new poll request resets the lease timer. So as long as the client polls at least as often as the lease time, the server SHOULD maintain the Watch. The following requests SHOULD reset the lease timer: read of the Watch URI itself or invocation of the `add`, `remove`, `pollChanges`, or `pollRefresh` operations.

Clients may request a difference lease time by writing to the `lease` object (requires servers to assign an href to the `lease` child). The server is free to honor the request, cap the lease within a specific range, or ignore the request. In all cases the write request will return a response containing the new lease time in effect.

Servers SHOULD report expired watches by returning an `err` object with the `BadUriErr` contract. As a general principle servers SHOULD honor watches until the lease runs out or the client explicitly invokes `delete`. However, servers are free to cancel watches as needed (such as power failure) and the burden is on clients to re-establish a new watch.

13.2.6 Watch.delete

The `delete` operation can be used to cancel an existing watch. Clients SHOULD always delete their watch when possible to be good oBIX citizens. However servers MUST always cleanup correctly without an explicit delete when the lease expires.

13.3 Watch Depth

When a watch is put on an object which itself has children objects, how does a client know how "deep" the subscription goes? oBIX requires watch depth to match an object's extent (see Section 10.3). When a watch is put on a target object, a server MUST notify the client of any changes to any of the objects within that target object's extent. If the extent includes `feed` objects they are not included in the watch – feeds have special watch semantics discussed in Section 13.4. This means a watch is inclusive of all descendents within the extent except `refs` and `feeds`.

13.4 Feeds

Servers may expose event streams using the `feed` object. The event instances are typed via the feed's `of` attribute. Clients subscribe to events by adding the feed's href to a watch, optionally passing an input parameter which is typed via the feed's `in` attribute. The object returned from `Watch.add` is a list of historic events (or the empty list if no event history is available). Subsequent calls to `pollChanges` returns the list of events which have occurred since the last poll.

Let's consider a simple example for an object which fires an event when its geographic location changes:

```
<obj href="/car/">
  <feed href="moved" of="/def/Coordinate"/>
<obj>
```

```
<obj href="/def/Coordinate">
  <real name="lat"/>
  <real name="long"/>
</obj>
```

We subscribe to the moved event feed by adding “/car/moved” to a watch. The WatchOut will include the list of any historic events which have occurred up to this point in time. If the server does not maintain an event history this list will be empty:

```
<obj is="obix:WatchIn">
  <list names="hrefs">
    <uri val="/car/moved" />
  </list>
</obj>

<obj is="obix:WatchOut">
  <list names="values">
    <feed href="/car/moved" of="/def/Coordinate/" />  <!-- empty history -->
  </list>
</obj>
```

Now every time we call `pollChanges` for the watch, the server will send us the list of event instances which have accumulated since our last poll:

```
<obj is="obix:WatchOut">
  <list names="values">
    <feed href="/car/moved" of="/def/Coordinate">
      <obj>
        <real name="lat"  val="37.645022"/>
        <real name="long" val="-77.575851"/>
      </obj>
      <obj>
        <real name="lat"  val="37.639046"/>
        <real name="long" val="-77.61872"/>
      </obj>
    </feed>
  </list>
</obj>
```

Note the feed’s `of` attribute works just like the `list`’s `of` attribute. The children event instances are assumed to inherit the contract defined by `of` unless explicitly overridden. If an event instance does override the `of` contract, then it MUST be contract compatible. Refer to the rules defined in Section 6.8.

Invoking a `pollRefresh` operation on a watch with a feed that has an event history, SHOULD return all the historical events as if the `pollRefresh` was an `add` operation. If an event history is not available, then `pollRefresh` SHOULD act like a normal `pollChanges` and just return the events which have occurred since the last poll.

14 Points

Anyone familiar with automation systems immediately identifies with the term *point* (sometimes called *tags* in the industrial space). Although there are many different definitions, generally points map directly to a sensor or actuator (called *hard points*). Sometimes the concept of a point is mapped to a configuration variable such as a software setpoint (called *soft points*). In some systems point is an atomic value, and in others it encapsulates a whole truckload of status and configuration information.

The goal of oBIX is to capture a normalization representation of points without forcing an impedance mismatch on vendors trying to make their native system oBIX accessible. To meet this requirement, oBIX defines a low level abstraction for point - simply one of the primitive value types with associated status information. Point is basically just a marker contract used to tag an object as exhibiting “point” semantics:

- `<obj href="obix:Point"/>`

This contract MUST only be used with the value primitive types: `bool`, `real`, `enum`, `str`, `abstime`, and `reltime`. Points SHOULD use the `status` attribute to convey quality information. The following table specifies how to map common control system semantics to a value type:

`bool`	digital point	`<bool is="obix:Point" val="true"/>`
`real`	analog point	`<real is="obix:Point" val="22" unit="obix:units/celsius"/>`
`enum`	multi-state point	`<enum is="obix:Point" val="slow"/>`

14.1 Writable Points

Different control systems handle point writes using a wide variety of semantics. Sometimes we write a point at a specific priority level. Sometimes we override a point for a limited period of time, after which the point falls back to a default value. The oBIX specification doesn’t attempt to impose a specific model on vendors. Rather oBIX provides a standard `WritablePoint` contract which may be extended with additional mixins to handle special cases. `WritablePoint` defines write as an operation which takes a `WritePointIn` structure containing the value to write. The contracts are:

```
<obj href="obix:WritablePoint" is="obix:Point">
  <op name="writePoint" in="obix:WritePointIn" out="obix:Point"/>
</obj>

<obj href="obix:WritePointIn">
  <obj name="value"/>
</obj>
```

It is implied that the value passed to `writePoint` match the type of the point. For example if `WritablePoint` is used with an `enum`, then `writePoint` MUST pass an `enum` for the value.

15 History

Most automation systems have the ability to persist periodic samples of point data to create a historical archive of a point's value over time. This feature goes by many names including logs, trends, or histories. In oBIX, a *history* is defined as a list of time stamped point values. The following features are provided by oBIX histories:

- **History Object**: a normalized representation for a history itself;
- **History Record**: a record of a point sampling at a specific timestamp
- **History Query**: a standard way to query history data as Points;
- **History Rollup**: a standard mechanism to do basic rollups of history data;
- **History Append**: ability to push new history records into a history;

15.1 History Object

Any object which wishes to expose itself as a standard oBIX history implements the `obix:History` contract:

```
<obj href="obix:History">
  <int     name="count"  min="0" val="0"/>
  <abstime name="start"  null="true"/>
  <abstime name="end"    null="true"/>
  <str     name="tz"     null="true"/>
  <op      name="query"  in="obix:HistoryFilter" out="obix:HistoryQueryOut"/>
  <feed    name="feed"   in="obix:HistoryFilter" of="obix:HistoryRecord"/>
  <op      name="rollup" in="obix:HistoryRollupIn" out="obix:HistoryRollupOut"/>
  <op      name="append" in="obix:HistoryAppendIn" out="obix:HistoryAppendOut"/>
</obj>
```

Let's look at each of `History's` sub-objects:

- **`count`**: this field stores the number of history records contained by the history;
- **`start`**: this field provides the timestamp of the oldest record. The timezone of this abstime MUST match `History.tz`;
- **`end`**: this field provides the timestamp of the newest record; The timezone of this abstime MUST match `History.tz`;
- **`tz`**: standardized timezone identifier for the history data (see 4.18.9)
- **`query`**: the query object is used to query the history to read history records;
- **`feed`**: used to subscribe to a real-time feed of history records;
- **`rollup`**: this object is used to perform history rollups (it is only supported for numeric history data);
- **`append`**: operation used to push new history records into the history

An example of a history which contains an hour of 15 minute temperature data:

```
<obj href="http://x/outsideAirTemp/history/" is="obix:History">
  <int     name="count"  val="5"/>
  <abstime name="start"  val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"    val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <str     name="tz"     val="America/New_York"/>
  <op      name="query"  href="query"/>
  <op      name="rollup" href="rollup"/>
</obj>
```

15.2 History Queries

Every `History` object contains a `query` operation to query the historical data.

15.2.1 HistoryFilter

The `History.query` input contract:

```
<obj href="obix:HistoryFilter">
  <int     name="limit"  null=”true”/>
  <abstime name="start"  null="true"/>
  <abstime name="end"    null="true"/>
</obj>
```

These fields are described in detail:

- **`limit`**: an integer indicating the maximum number of records to return. Clients can use this field to throttle the amount of data returned by making it non-null. Servers MUST never return more records than the specified limit. However servers are free to return fewer records than the limit.
- **`start`**: if non-null this field indicates an inclusive lower bound for the query's time range. This value SHOULD match the history's timezone, otherwise the server MUST normalize based on absolute time.
- **`end`**: if non-null this field indicates an inclusive upper bound for the query's time range. This value SHOULD match the history's timezone, otherwise the server MUST normalize based on absolute time.

15.2.2 HistoryQueryOut

The `History.query` output contract:

```
<obj href="obix:HistoryQueryOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list    name="data"  of="obix:HistoryRecord"/>
</obj>
```

Just like `History`, every `HistoryQueryOut` returns `count`, `start`, and `end`. But unlike `History`, these values are for the query result, not the entire history. The actual history data is stored as a list of `HistoryRecords` in the `data` field. Remember that child order is not guaranteed in oBIX, therefore it might be common to have `count` after `data`. The start, end, and data HistoryRecord timestamps MUST have a timezone which matches `History.tz`.

15.2.3 HistoryRecord

The `HistoryRecord` contract specifies a record in a history query result:

```
<obj href="obix:HistoryRecord">
  <abstime name="timestamp" null="true"/>
  <obj     name="value"     null="true"/>
</obj>
```

Typically the value SHOULD be one of the value types used with `obix:Point`.

15.2.4 History Query Example

An example query from the "/outsideAirTemp/history" example above:

```
<obj href="http://x/outsideAirTemp/history/query" is="obix:HistoryQueryOut">
  <int name="count" val="5">
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"   val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <list name="data" of="#RecordDef obix:HistoryRecord">
    <obj> <abstime name="timestamp" val="2005-03-16T14:00:00-05:00"/>
          <real name="value" val="40"/> </obj>
    <obj> <abstime name="timestamp"  val="2005-03-16T14:15:00-05:00"/>
```

```
          <real name="value" val="42"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:30:00-05:00"/>
          <real name="value" val="43"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:45:00-05:00"/>
          <real name="value" val="47"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T15:00:00-05:00"/>
          <real name="value" val="44"/> </obj>
  </list>
  <obj href="#RecordDef" is="obix:HistoryRecord">
    <abstime name="timestamp" tz="America/New_York"/>
    <real name="value" unit="obix:units/fahrenheit"/>
  </obj>
</obj>
```

Note in the example above how the `data` list uses a document local contract to define facets common to all the records (although we still have to flatten the contract list).

15.3 History Rollups

Control systems collect historical data as raw time sampled values. However, most applications wish to consume historical data in a summarized form which we call *rollups*. The rollup operation is used to summarize an interval of time. History rollups only apply to histories which store numeric information as a list of `RealPoints`. Attempting to query a rollup on a non-numeric history such as a history of `BoolPoints` SHOULD result in an error.

15.3.1 HistoryRollupIn

The `History.rollup` input contract extends `HistoryFilter` to add an interval parameter:

```
<obj href="obix:HistoryRollupIn" is="obix:HistoryFilter">
  <reltime name="interval"/>
</obj>
```

15.3.2 HistoryRollupOut

The `History.rollup` output contract:

```
<obj href="obix:HistoryRollupOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list name="data" of="obix:HistoryRollupRecord"/>
</obj>
```

The `HistoryRollupOut` object looks very much like `HistoryQueryOut` except it returns a list of `HistoryRollupRecords`, rather than `HistoryRecords`. Note: unlike `HistoryQueryOut`, the `start` for `HistoryRollupOut` is exclusive, not inclusive. This issue is discussed in greater detail next. The start, end, and data HistoryRollupRecord timestamps MUST have a timezone which matches `History.tz`.

15.3.3 HistoryRollupRecord

A history rollup returns a list of `HistoryRollupRecords`:

```
<obj href="obix:HistoryRollupRecord">
  <abstime name="start"/>
  <abstime name="end"  />
  <int  name="count"/>
  <real name="min" />
  <real name="max" />
  <real name="avg" />
  <real name="sum" />
</obj>
```

The children are defined as:

- **`start`**: the exclusive start time of the record's rollup interval;

- **end**: the inclusive end time of the record's rollup interval;
- **count**: the number of records used to compute this rollup interval;
- **min**: specifies the minimum value of all the records within the interval;
- **max**: specifies the maximum value of all the records within the interval;
- **avg**: specifies the mathematical average of all the values within the interval;
- **sum**: specifies the summation of all the values within the interval;

15.3.4 Rollup Calculation

The best way to understand how rollup calculations work is through an example. Let's consider a history of meter data where we collected two hours of 15 minute readings of kilowatt values:

```
<obj is="obix:HistoryQueryOut">
  <int     name="count" val="9">
  <abstime name="start" val="2005-03-16T12:00:00+04:00" tz="Asia/Dubai"/>
  <abstime name="end"   val="2005-03-16T14:00:00+04:00" tz="Asia/Dubai"/>
  <list name="data" of="#HistoryDef obix:HistoryRecord">
    <obj> <abstime name="timestamp" val="2005-03-16T12:00:00+04:00"/>
          <real name="value" val="80"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:15:00+04:00"/>
          <real name="value" val="82"></obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:30:00+04:00"/>
          <real name="value" val="90"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:45:00+04:00"/>
          <real name="value" val="85"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:00:00+04:00"/>
          <real name="value" val="81"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:15:00+04:00"/>
          <real name="value" val="84"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:30:00+04:00"/>
          <real name="value" val="91"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:45:00+04:00"/>
          <real name="value" val="83"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:00:00+04:00"/>
          <real name="value" val="78"> </obj>
  </list>
  <obj href="#HistoryRecord" is="obix:HistoryRecord">
    <abstime name="timestamp" tz="Asia/Dubai"/>
    <real name="value" unit="obix:units/kilowatt"/>
  <obj>
</obj>
```

If we were to query the rollup using an interval of 1 hour with a start time of 12:00 and end time of 14:00, the result should be:

```
<obj is="obix:HistoryRollupOut obix:HistoryQueryOut">
  <int     name="count" val="2">
  <abstime name="start" val="2005-03-16T12:00:00+04:00 tz="Asia/Dubai"/>
  <abstime name="end"   val="2005-03-16T14:00:00+04:00" tz="Asia/Dubai"/>
  <list name="data" of="obix:HistoryRollupRecord">
    <obj>
      <abstime name="start" val="2005-03-16T12:00:00+04:00"
               tz="Asia/Dubai"/>
      <abstime name="end"   val="2005-03-16T13:00:00+04:00"
               tz="Asia/Dubai"/>
      <int  name="count" val="4"    />
      <real name="min"   val="81"   />
      <real name="max"   val="90"   />
      <real name="avg"   val="84.5" />
      <real name="sum"   val="338"  />
    </obj>
    <obj>
      <abstime name="start" val="2005-03-16T13:00:00+04:00"
               tz="Asia/Dubai"/>
      <abstime name="end"   val="2005-03-16T14:00:00+04:00"
               tz="Asia/Dubai"/>
      <int  name="count" val="4"   />
      <real name="min"   val="78"  />
```

```
      <real name="max"   val="91"  />
      <real name="avg"   val="84"  />
      <real name="sum"   val="336" />
    </obj>
  </list>
</obj>
```

If you whip out your calculator, the first thing you will note is that the first raw record of 80kW was never used in the rollup. This is because start time is always exclusive. The reason start time has to be exclusive is because we are summarizing discrete samples into a contiguous time range. It would be incorrect to include a record in two different rollup intervals! To avoid this problem we always make start time exclusive and end time inclusive. The following table illustrates how the raw records were applied to rollup intervals:

Interval Start (exclusive)	Interval End (inclusive)	Records Included
2005-03-16T12:00	2005-03-16T13:00	82 + 90 + 85 + 81 = 338
2005-03-16T13:00	2005-03-16T14:00	84 + 91 + 83 + 78 = 336

15.4 History Feeds

The `History` contract specifies a feed for subscribing to a real-time feed of the history records. `History.feed` reuses the same `HistoryFilter` input contract used by `History.query` – the same semantics apply. When adding a History feed to a watch, the initial result SHOULD contain the list of `HistoryRecords` filtered by the input parameter (the initial result should match what `History.query` would return). Subsequent calls to `Watch.pollChanges` SHOULD return any new `HistoryRecords` which have been collected since the last poll that also satisfy the `HistoryFilter`.

15.5 History Append

The `History.append` operation allows a client to push new HistoryRecords into a History log (assuming proper security credentials). This operation comes in handy when bi-direction HTTP connectivity is not available. For example if a device in the field is behind a firewall, it can still push history data on an interval basis to a server using the append operation.

15.5.1 HistoryAppendIn

The `History.append` input contract:

```
<obj href="obix:HistoryAppendIn">
  <list name="data" of="obix:HistoryRecord"/>
</obj>
```

The `HistoryAppendIn` is a wrapper for the list of HistoryRecords to be inserted into the History. The HistoryRecords SHOULD use a timestamp which matches `History.tz`. If the timezone doesn't match, then the server MUST normalize to its configured timezone based on absolute time. The HistoryRecords in the data list MUST be sorted by timestamp from oldest to newest, and MUST not include a timestamp equal to or older than `History.end`.

15.5.2 HistoryAppendOut

The `History.append` output contract:

```
<obj href="obix:HistoryAppendOut">
  <int     name="numAdded"/>
  <int     name="newCount"/>
  <abstime name="newStart" null="true"/>
  <abstime name="newEnd"   null="true"/>
</obj>
```

The output of the append operation returns the number of new records appended to the History and the new total count, start time, and end time of the entire History. The newStart and newEnd timestamps MUST have a timezone which matches `History.tz`.

16 Alarming

The oBIX alarming feature specifies a normalized model to query, watch, and acknowledge alarms. In oBIX, an alarm indicates a condition which requires notification of either a user or another application. In many cases an alarm requires acknowledgement, indicating that someone (or something) has taken action to resolve the alarm condition. The typical lifecycle of an alarm is:

1. **Source Monitoring**: algorithms in a server monitor an *alarm source*. An alarm source is an object with an href which has the potential to generate an alarm. Example of alarm sources might include sensor points (this room is too hot), hardware problems (disk is full), or applications (building is consuming too much energy at current energy rates)
2. **Alarm Generation**: if the algorithms in the server detect that an alarm source has entered an alarm condition, then an *alarm* record is generated. Every alarm is uniquely identified using an href and represented using the `obix:Alarm` contract. Sometimes we refer to the alarm transition as *off-normal*.
3. **To Normal**: many alarm sources are said to be *stateful* - eventually the alarm source exits the alarm state, and is said to return *to-normal*. Stateful alarms implement the `obix:StatefulAlarm` contract. When the source transitions to normal, we update `normalTimestamp` of the alarm.
4. **Acknowledgement**: often we require that a user or application acknowledges that they have processed an alarm. These alarms implement the `obix:AckAlarm` contract. When the alarm is acknowledged, we update `ackTimestamp` and `ackUser`.

16.1 Alarm States

Alarm state is summarized with two variables:

- **In Alarm**: is the alarm source currently in the alarm condition or in the normal condition. This variable maps to the `alarm` status state.
- **Acknowledged**: is the alarm acknowledged or unacknowledged. This variable maps to the `unacked` status state.

Either of these states may transition independent of the other. For example an alarm source can return to normal before or after an alarm has been acknowledged. Furthermore it is not uncommon to transition between normal and off-normal multiple times generating several alarm records before any acknowledgements occur.

Note not all alarms have state. An alarm which implements neither `StatefulAlarm` nor the `AckAlarm` contracts is completely stateless – these alarms merely represent event. An alarm which implements `StatefulAlarm` but not `AckAlarm` will have an in-alarm state, but not acknowledgement state. Conversely an alarm which implements `AckAlarm` but not `StatefulAlarm` will have an acknowledgement state, but not in-alarm state.

16.1.1 Alarm Source

The current alarm state of an alarm source is represented using the `status` attribute. This attribute is discussed in Section 4.18.8. It is recommended that alarm sources always report their status via the `status` attribute.

16.1.2 StatefulAlarm and AckAlarm

An `Alarm` record is used to summarize the entire lifecycle of an alarm event. If the alarm implements `StatefulAlarm` it tracks transition from off-normal back to normal. If the alarm implements `AckAlarm`, then it also summarizes the acknowledgement. This allows for four discrete alarm states:

alarm	acked	normalTimestamp	ackTimestamp
true	false	null	null
true	true	null	non-null
false	false	non-null	null
false	true	non-null	non-null

16.2 Alarm Contracts

16.2.1 Alarm

The core `Alarm` contract is:

```
<obj href="obix:Alarm">
  <ref name="source"/>
  <abstime name="timestamp"/>
</obj>
```

The child objects are:

- **source**: the URI which identifies the alarm source. The source SHOULD reference an oBIX object which models the entity that generated the alarm.
- **timestamp**: this is the time at which the alarm source transitioned from normal to off-normal and the Alarm record was created.

16.2.2 StatefulAlarm

Alarms which represent an alarm state which may transition back to normal SHOULD implement the `StatefulAlarm` contract:

```
<obj href="obix:StatefulAlarm" is="obix:Alarm">
  <abstime name="normalTimestamp" null="true"/>
</obj>
```

The child object is:

- **normalTimestamp**: if the alarm source is still in the alarm condition, then this field is null. Otherwise this indicates the time of the transition back to the normal condition.

16.2.3 AckAlarm

Alarms which support acknowledgement SHOULD implement the `AckAlarm` contract:

```
<obj href="obix:AckAlarm" is="obix:Alarm">
  <abstime name="ackTimestamp" null="true"/>
  <str name="ackUser" null="true"/>
  <op name="ack" in="obix:AlarmAckIn" out="obix:AlarmAckOut"/>
</obj>

<obj href="obix:AckAlarmIn">
  <str name="ackUser" null="true"/>
</obj>

<obj href="obix:AckAlarmOut">
  <obj name="alarm" is="obix:AckAlarm obix:Alarm"/>
</obj>
```

The child objects are:

- **ackTimestamp**: if the alarm is unacknowledged, then this field is null. Otherwise this indicates the time of the acknowledgement.
- **ackUser**: if the alarm is unacknowledged, then this field is null. Otherwise this field should provide a string indicating who was responsible for the acknowledgement.

The `ack` operation is used to programmatically acknowledge the alarm. The client may optionally specify an `ackUser` string via `AlarmAckIn`. However, the server is free to ignore this field depending on security conditions. For example a highly trusted client may be allowed to specify its own `ackUser`, but a less trustworthy client may have its `ackUser` predefined based on the authentication credentials of the protocol binding. The `ack` operation returns an `AckAlarmOut` which contains the updated alarm record. Use the `Lobby.batch` operation to efficiently acknowledge a set of alarms.

16.2.4 PointAlarms

It is very common for an alarm source to be an `obix:Point`. A respective `PointAlarm` contract is provided as a normalized way to report the value which caused the alarm condition:

```
<obj href="obix:PointAlarm" is="obix:Alarm">
  <obj name="alarmValue"/>
</obj>
```

The `alarmValue` object SHOULD be one of the value types defined for `obix:Point` in Section 14.

16.3 AlarmSubject

Servers which implement oBIX alarming MUST provide one or more objects which implement the `AlarmSubject` contract. The `AlarmSubject` contract provides the ability to categorize and group the sets of alarms a client may discover, query, and watch. For instance a server could provide one `AlarmSubject` for all alarms and other `AlarmSubjects` based on priority or time of day. The contract for `AlarmSubject` is:

```
<obj href="obix:AlarmSubject">
  <int     name="count"  min="0" val="0"/>
  <op      name="query" in="obix:AlarmFilter" out="obix:AlarmQueryOut"/>
  <feed    name="feed"  in="obix:AlarmFilter" of="obix:Alarm"/>
</obj>

<obj href="obix:AlarmFilter">
  <int     name="limit"  null="true"/>
  <abstime name="start"  null="true"/>
  <abstime name="end"    null="true"/>
</obj>

<obj href="obix:AlarmQueryOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list    name="data"  of="obix:Alarm"/>
</obj>
```

The `AlarmSubject` follows the same design pattern as `History`. The `AlarmSubject` specifies the active `count` of alarms; however, unlike `History` it does not provide the `start` and `end` bounding timestamps. It contains a `query` operation to read the current list of alarms with an `AlarmFilter` to filter by time bounds. `AlarmSubject` also contains a feed object which may be used to subscribe to the alarm events.

16.4 Alarm Feed Example

The following example illustrates how a feed works with this `AlarmSubject`:

```
<obj is="obix:AlarmSubject" href="/alarms/">
  <int  name="count" val="2"/>
  <op   name="query" href="query"/>
```

```
  <feed name="feed"  href="feed" />
</obj>
The server indicates it has two open alarms under the specified AlarmSubject. If a client
were to add the AlarmSubject’s feed to a watch:
<obj is="obix:WatchIn">
 <list names="hrefs"/>
  <uri val="/alarms/feed">
    <obj name="in" is="obix:AlarmFilter">
      <int name="limit" val="25"/>
    </obj>
  </uri>
 </list>
</obj>

<obj is="obix:WatchOut">
 <list names="values">
   <feed href="/alarms/feed" of="obix:Alarm">
    <obj href="/alarmdb/528" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/airHandlers/2/returnTemp"/>
      <abstime name="timestamp" val="2006-05-18T14:20:00Z"/>
      <abstime name="normalTimestamp"  null="null"/>
      <real name="alarmValue" val="80.2"/>
    </obj>
    <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/doors/frontDoor"/>
      <abstime name="timestamp" val="2006-05-18T14:18:00Z"/>
      <abstime name=" normalTimestamp" null="null"/>
      <real name="alarmValue" val="true"/>
    </obj>
  </feed>
 </list>
</obj>
```

The watch returns the historic list of alarm events which is two open alarms. The first alarm indicates an out of bounds condition in AirHandler-2’s return temperature. The second alarm indicates that the system has detected that the front door has been propped open.

Now let’s fictionalize that the system detects the front door is closed, and alarm point transitions to the normal state. The next time the client polls the watch the alarm would show up in the feed list (along with any additional changes or new alarms not shown here):

```
<obj is="obix:WatchOut">
 <list names="values">
   <feed href="/alarms/feed" of="obix:Alarm">>
    <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/doors/frontDoor"/>
      <abstime name="timestamp" val="2006-05-18T14:18:00Z"/>
      <abstime name=" normalTimestamp" null="2006-05-18T14:45:00Z"/>
      <real name="alarmValue" val="true"/>
    </obj>
   </feed>
 </list>
</obj>
```

17 Security

Security is a broad topic, that covers many issues:

- **Authentication**: verifying a user (client) is who he says he is;
- **Encryption**: protecting oBIX documents from prying eyes;
- **Permissions**: checking a user's permissions before granting access to read/write objects or invoke operations;
- **User Management**: managing user accounts and permissions levels;

The basic philosophy of oBIX is to leave these issues outside of the specification. Authentication and encryption is left as a protocol binding issue. Privileges and user management is left as a vendor implementation issue. Although it is entirely possible to define a publicly exposed user management model through oBIX, this specification does not define any standard contracts for user management.

17.1 Error Handling

It is expected that an oBIX server will perform authentication and utilize those user credentials for checking permissions before processing read, write, and invoke requests. As a general rule, servers SHOULD return `err` with the `obix:PermissionErr` contract to indicate a client lacks the permission to perform a request. In particularly sensitive applications, a server may instead choose to return `BadUriErr` so that an untrustworthy client is unaware that a specific object even exists.

17.2 Permission based Degradation

Servers SHOULD strive to present their object model to a client based on the privileges available to the client. This behavior is called *permission based degradation*. The following rules summarize effective permission based degradation:

1. If an object cannot be read, then it SHOULD NOT be discoverable through objects which are available.
2. Servers SHOULD attempt to group standard contracts within the same privilege level – for example don't split `obix:History`'s `start` and `end` into two different security levels such that a client might be able to read `start`, and not `end`.
3. Servers SHOULD NOT include a contract in an object's `is` attribute if the contract's children are not readable to the client.
4. If an object isn't writable, then the `writable` attribute SHOULD be set to `false` (either explicitly or through a contract default).
5. If an `op` inherited from a visible contract cannot be invoked, then the server SHOULD set the `null` attribute to `true` to disable it.

18 HTTP Binding

The HTTP binding specifies a simple REST mapping of oBIX requests to HTTP. A read request is a simple HTTP GET, which means that you can simply read an object by typing its URI into your browser. Refer to “RFC 2616 Hypertext Transfer Protocol” for the full specification of HTTP 1.1.

18.1 Requests

The following table summarizes how oBIX requests map to HTTP methods:

oBIX Request	HTTP Method	Target
Read	GET	Any object with an href
Write	PUT	Any object with an href and writable=true
Invoke	POST	Any op object

The URI used for an HTTP request MUST map to the URI of the object being read, written, or invoked. Read requests use a simple HTTP GET and return the resulting oBIX document. Write and invoke are implemented with the PUT and POST methods respectively. The input is passed to the server as an oBIX document and the result is returned as an oBIX document.

If the oBIX server processes a request, then it MUST return the resulting oBIX document with an HTTP status code of 200 OK. The 200 status code MUST be used even if the request failed and the server is returning an `err` object as the result.

18.2 MIME Type

If XML encoding is used, then the oBIX documents passed between client and servers SHOULD specify a MIME type of “text/xml” for the Content-Type HTTP header. Clients and servers MUST encode the oBIX document passed over the network using standard XML encoding rules. It is strongly RECOMMENDED to use UTF8 without a byte-order mark. If specified, the Content-Encoding HTTP header MUST match the XML encoding.

If the binary encoding is used, then the MIME type of “application/x-obix-binary” MUST be used.

18.3 Content Negotiation

oBIX resources may be encoded using either the “text/xml“ or the “application/x-obix-binary“ MIME types. Clients and servers SHOULD follow Section 12 of RFC 2616 for content negotiation.

If a client wishes to GET a resource using a specific encoding, then it SHOULD specify the desired MIME type in the Accept header.

If the server does not support the MIME type of a client request, then it SHOULD respond with the 406 Not Acceptable status code. There are two use cases for a 406 failure: 1) the client specifies an unsupported MIME type in the Accept header of a GET (read) request, or 2) the client specifies an unsupported MIME type in the Content-Type of a PUT (write) or POST (invoke) request.

18.4 Security

Numerous standards are designed to provide authentication and encryption services for HTTP. Existing standards SHOULD be used when applicable for oBIX HTTP implementations including:

- RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication
- RFC 2818 - HTTP Over TLS (HTTPS)
- RFC 4346/2246 – The TLS Protocol (Transport Layer Security)

18.5 Localization

Servers SHOULD localize appropriate data based on the desired locale of the client agent. Localization should include the `display` and `displayName` attributes. The desired locale of the client should be determined through authentication or via the Accept-Language HTTP header. A suggested algorithm is to check if the authenticated user has a preferred locale configured in the server's user database, and if not then fallback to the locale derived from the Accept-Language header.

Localization MAY include auto-conversion of units. For example if the authenticated user has a configured a preferred unit system such as English versus Metric, then the server might attempt to convert values with an associated `unit` facet to the desired unit system.

19 SOAP Binding

The SOAP binding maps a SOAP operation to each of the three oBIX request types: read, write and invoke. Like the HTTP binding, read is supported by every object, write is supported by objects whose `writable` attribute is `true`, and invoke is only supported by operations. Inputs and outputs of each request are specific to the target object.

Unlike the HTTP binding, requests are not accessed via the URI of the target object, but instead via the URI of the SOAP server with the object's URI encoded into the body of the SOAP envelope.

19.1 SOAP Example

The following is a SOAP request to an oBIX server's `About` object:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Body>
  <read xmlns="http://obix.org/ns/wsdl/1.1"
        href="http://localhost/obix/about" />
</env:Body>
</env:Envelope>
```

An example response to the above request:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Body>
  <obj name="about"
       href="http://localhost/obix/about/"
       xmlns="http://obix.org/ns/schema/1.1">
     <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>
 </env:Body>
</env:Envelope>
```

19.2 Error Handling

The oBIX specification defines no SOAP faults. If a request is processed by an oBIX server, then a valid oBIX document SHOULD be returned with a failure indicated via the `err` object.

19.3 Security

Refer to the recommendations in WS-I Basic Profile 1.0 for security:

http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html#security

19.4 Localization

SOAP bindings SHOULD follow localization patterns defined for the HTTP binding when applicable (see Section 18.5).

19.5 WSDL

In the types section of the WSDL document, the oBIX schema is imported. Server implementations might consider providing the `schemaLocation` attribute which is absent in the standard document.

Missing from the standard oBIX WSDL is the service element. This element binds a SOAP server instance with a network address. Each instance will have to provide its own services section of the WSDL document. The following is an example of the WSDL service element:

```
<wsdl:service name="obix">
  <wsdl:port name="obixPort" binding="tns:obixSoapBinding">
    <soap:address location="http://localhost/obix/soap"/>
  </wsdl:port>
</wsdl:service>
```

Standard oBIX WSDL is:

```
<wsdl:definitions targetNamespace="http://obix.org/ns/wsdl/1.1"
                  xmlns="http://obix.org/ns/wsdl/1.1"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:obix="http://obix.org/ns/schema/1.1">
 <wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              elementFormDefault="qualified"
              targetNamespace="http://obix.org/ns/wsdl/1.1">
   <xsd:import namespace="http://obix.org/ns/schema/1.1"/>
   <xsd:complexType name="ReadReq">
    <xsd:attribute name="href" type="xsd:anyURI"/>
   </xsd:complexType>
   <xsd:complexType name="WriteReq">
    <xsd:complexContent>
     <xsd:extension base="ReadReq">
      <xsd:sequence>
       <xsd:element ref="obix:obj" maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
     </xsd:extension>
    </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="InvokeReq">
    <xsd:complexContent>
     <xsd:extension base="ReadReq">
      <xsd:sequence>
       <xsd:element ref="obix:obj" maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
     </xsd:extension>
    </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="read" type="ReadReq"/>
   <xsd:element name="write" type="WriteReq"/>
   <xsd:element name="invoke" type="InvokeReq"/>
  </xsd:schema>
 </wsdl:types>
 <wsdl:message name="readSoapReq">
  <wsdl:part name="body" element="read"/>
 </wsdl:message>
 <wsdl:message name="readSoapRes">
  <wsdl:part name="body" element="obix:obj"/>
 </wsdl:message>
 <wsdl:message name="writeSoapReq">
  <wsdl:part name="body" element="write"/>
 </wsdl:message>
 <wsdl:message name="writeSoapRes">
  <wsdl:part name="body" element="obix:obj"/>
 </wsdl:message>
 <wsdl:message name="invokeSoapReq">
  <wsdl:part name="body" element="invoke"/>
 </wsdl:message>
 <wsdl:message name="invokeSoapRes">
  <wsdl:part name="body" element="obix:obj"/>
```

```
  </wsdl:message>
  <wsdl:portType name="oBIXSoapPort">
   <wsdl:operation name="read">
    <wsdl:input message="readSoapReq"/>
    <wsdl:output message="readSoapRes"/>
   </wsdl:operation>
   <wsdl:operation name="write">
    <wsdl:input message="writeSoapReq"/>
    <wsdl:output message="writeSoapRes"/>
   </wsdl:operation>
   <wsdl:operation name="invoke">
    <wsdl:input message="invokeSoapReq"/>
    <wsdl:output message="invokeSoapRes"/>
   </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="oBIXSoapBinding" type="oBIXSoapPort">
   <soap:binding style="document"
                 transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="read">
    <soap:operation soapAction="http://obix.org/ns/wsdl/1.1/read"
                    style="document"/>
    <wsdl:input>
     <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="write">
    <soap:operation soapAction="http://obix.org/ns/wsdl/1.1/write"
                    style="document"/>
    <wsdl:input>
     <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="invoke">
    <soap:operation soapAction="http://obix.org/ns/wsdl/1.1/invoke"
                    style="document"/>
    <wsdl:input>
     <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

20 Conformance

The last numbered section in the specification must be the Conformance section. Conformance Statements/Clauses go here.

Acknowledgements

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*
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, Tridium, Inc.
Craig Gemmill, Tridium, Inc.
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)*
Rob Zivney, Hirsch Electronics Corporation

A. Appendices

No-normative and explanatory information goes in the appendices.

2698 # Revision History

Rev	Date	By Whom	What
wd-0.1	14 Jan 03	Brian Frank	Initial version
wd-0.2	22 Jan 03	Brian Frank	
wd-0.3	30 Aug 04	Brian Frank	Move to Oasis, SysService
wd-0.4	2 Sep 04	Brian Frank	Status
wd-0.5	12 Oct 04	Brian Frank	Namespaces, Writes, Poll
wd-0.6	2 Dec 04	Brian Frank	Incorporate schema comments
wd-0.7	17 Mar 05	Brian Frank	URI, REST, Prototypes, History
wd-0.8	19 Dec 05	Brian Frank	Contracts, Ops
wd-0.9	8 Feb 06	Brian Frank	Watches, Alarming, Bindings
wd-0.10	13 Mar 06	Brian Frank	Overview, XML, clarifications
wd-0.11	20 Apr 06	Brian Frank	10.1 sections, ack, min/max
wd-0.11.1	28 Apr 06	Aaron Hansen	WSDL Corrections
wd-0.12	22 May 06	Brian Frank	Status, feeds, no deltas
wd-0.12.1	29 Jun 06	Brian Frank	Schema, stdlib corrections
obix-1.0-cd-02	30 Jun 06	Aaron Hansen	OASIS document format compliance.
obix-1.0-cs-01	18 Oct 06	Brian Frank	Public review comments
wd-obix.1.1.1	26 Nov 07	Brian Frank	Fixes, date, time, tz
wd-obix.1.1.2	11 Nov 08	Craig Gemmill (from Aaron Hansen)	Add iCalendar scheduling
wd-obix-1.1.3	10 Oct 09	Brian Frank	Remove Scheduling chapter Rev namespace to 1.1 Add Binary Encoding chapter
wd-obix-1.1.4	12 Nov 09	Brian Frank	MUST, SHOULD, MAY History.tz, History.append HTTP Content Negotiation
oBIX-1-1-spec-wd05	01 Jun 10	Toby Considine	Updated to current OASIS Templates, requirements
oBIX-1-1-spec-wd06	08 Jun 10	Brad Benson	Custom facets within binary encoding