Extensible Presentation Language (XPL) and
Type Resolution Protocol (XPL/TRP)
draft-ryanpitt-6lowapp-xpl-00

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on April 22, 2010, at which time copyright will revert to the persons named herein as authors.

Comments are solicited and should be addressed to the authors.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

This document proposes an Extensible Presentation Language (XPL) and XPL Type Resolution Protocol to be used as the basis for building protocols in the application area addressed by the IETF 6lowpan and proposed 6lowApp Working Groups. This comprises applications for highly resource-constrained devices over wireless networks, where the devices are: power-constrained; ubiquitous; potentially very small; heterogeneous; have very short duty cycles (e.g. 0.1%); and are installed or used for many years on end, e.g. ‘smart’ electricity meters, electric cars, etc.
Table of Contents

1. Introduction ................................................. 5
   1.1. Components of the proposal ............................. 6
   1.2. Benefits ............................................ 6
   1.3. Potted History ....................................... 7
   1.4. XPL Introduction .................................... 9
2. Basic Concepts .............................................. 11
   2.1. Type Library Locations ................................. 12
   2.2. Type Maps ........................................... 13
3. Extensible Presentation Language .............................. 15
   3.1. Metadata ............................................ 15
      3.1.1. XPL Text Format Grammar .......................... 16
      3.1.2. Primitive Types ................................ 16
      3.1.3. Compound Types ................................ 16
      3.1.4. Array Types ..................................... 17
      3.1.5. Reference Identifiers ............................ 17
   3.2. Type Library Structure ................................. 17
      3.2.1. Type Library Entries ............................... 18
      3.2.2. Type Library Locations ............................ 18
         3.2.2.1. library.base ................................ 18
         3.2.2.2. library.name ................................ 19
         3.2.2.3. library.definition ............................ 19
         3.2.2.4. library.relation .............................. 19
      3.3. XPL Metadata Definitions ............................. 20
         3.3.1. Primitive Types ................................. 20
         3.3.2. References .................................... 21
         3.3.3. Compound Types ................................ 21
         3.3.4. Arrays .......................................... 21
         3.3.5. Sequence ........................................ 22
         3.3.6. Tag ............................................. 23
         3.3.7. Encoding ........................................ 23
         3.3.8. Envelope ........................................ 24
         3.3.9. Abstract ........................................ 24
      3.4. Meta Dictionary Description .......................... 26
      3.5. Library Types ....................................... 28
         3.5.1. library.base .................................... 28
         3.5.2. library.name .................................... 28
         3.5.3. library.definition ............................... 29
         3.5.4. library.relation ................................. 29
         3.5.5. library.entry ................................... 29
         3.5.6. library.list .................................... 30
      3.6. Meta Dictionary Reference ............................ 30
         3.6.1. library.base .................................... 30
         3.6.2. uint8 ........................................... 30
         3.6.3. uvarint28 ........................................ 30
         3.6.4. meta ............................................. 31
         3.6.5. meta.id .......................................... 31
3.6.6. meta.cluster .......................... 31
3.6.7. meta.abstract_map .......................... 31
3.6.8. meta.abstract .......................... 32
3.6.9. u8utf8 ................................ 32
3.6.10. meta.name .......................... 32
3.6.11. meta.version .......................... 33
3.6.12. meta.definition .......................... 33
3.6.13. meta.expression .......................... 33
3.6.14. meta.reference .......................... 34
3.6.15. meta.tag .......................... 34
3.6.16. meta.sequence .......................... 34
3.6.17. meta.array .......................... 35
3.6.18. meta.envelope .......................... 35
3.6.19. meta.encoding .......................... 35
3.6.20. meta.atom .......................... 36
3.6.21. meta.atomic_attribute .......................... 36
3.6.22. meta.attribute .......................... 37
3.6.23. meta.attribute.size .......................... 37
3.6.24. meta.attribute.integer .......................... 37
3.6.25. meta.attribute.unsigned .......................... 37
3.6.26. meta.attribute.bigendian .......................... 37
3.6.27. dictionary .......................... 37
3.6.28. dictionary.base .......................... 38
3.6.29. dictionary.name .......................... 38
3.6.30. dictionary.definition .......................... 38
3.6.31. dictionary.relation .......................... 38
3.6.32. dictionary.location .......................... 39
3.6.33. dictionary.definition_envelope .......................... 39
3.6.34. dictionary.entry .......................... 39
3.6.35. dictionary.entry.list .......................... 40
3.7. Meta Dictionary Mappings .......................... 40
3.8. Meta Dictionary Encoding .......................... 42
4. XPL Type Resolution Protocol .......................... 43
4.1. Message Types .......................... 43
4.2. Protocol Definition .......................... 44
4.2.1. Request .......................... 44
4.2.2. Response .......................... 45
4.3. Check Core .......................... 46
4.3.1. Request .......................... 46
4.3.2. Response .......................... 46
4.4. Map Type .......................... 47
4.4.1. Request .......................... 47
4.4.2. Response .......................... 48
4.5. Map Default .......................... 48
4.5.1. Request .......................... 48
4.5.2. Response .......................... 49
4.6. Reserve Type .......................... 49
4.6.1. Request .......................... 50
4.6.2.  Response ............................................. 50
4.7.  Map Reverse ........................................... 50
   4.7.1.  Request ........................................... 50
   4.7.2.  Response ........................................... 51
4.8.  Message ................................................ 51
   4.8.1.  Request/Response .................................. 51
4.9.  Error .................................................. 51
   4.9.1.  Response ........................................... 52
5.  XPL Message Format ....................................... 52
   5.1.  Message Format Type Description .................. 53
6.  Acknowledgements ......................................... 53
7.  IANA Considerations ...................................... 53
8.  Security Considerations .................................. 53
9.  Informative References ................................... 53
Appendix A.  Additional Information ........................ 54
   A.1.  Exclusions .......................................... 54
   A.2.  Usual Suspects ....................................... 55
Authors’ Addresses ............................................. 56
Intellectual Property and Copyright Statements .............. 57
1. Introduction

The IETF 6lowpan and proposed 6lowApp Working Groups concern applications for highly resource-constrained devices over wireless networks, where the devices are power-constrained; ubiquitous; potentially very small; heterogeneous; have very short duty cycles (e.g. 0.1%); and are installed or used for many years on end, e.g. 'smart' electricity meters, electric cars, etc.

This application area presents extreme challenges in respect of processing power, storage, bandwidth, device lifetime, and by implication requires extreme compactness in protocol/schema/type definition, and extreme flexibility in protocol versioning.

As the concept of the "Internet of Things" becomes more of a reality and true ubiquitous computing evolves, a large and ever-increasing number of application protocols will be created. Many of these need to be standardised so a protocol description framework is implicitly required. While there is a short term need for the ZigBee Smart Energy application protocol to be defined, there will be many other standardised protocols created in the future.

Monolithic protocol-definition systems like CORBA & SOAP attempt to provide an end-to-end solution incorporating wire protocols, payload encapsulation, and software APIs. In contrast, the REST architectural style isn’t a protocol-definition system at all, but it sets a framework on which to build solutions. This architectural style requires an embodiment such as HTTP and the addition of other technologies such as XML, WSDL/WADL, and possibly extensive documentation to form a complete application protocol. These hybrid systems provide additional flexibility over monolithic approaches like CORBA at the cost of requiring additional documentation and possibly incorporating many external specifications.

In both the CORBA/SOAP and REST hybrid approach the information used to describe the protocol is removed from the application at runtime. This Draft takes a different approach to both, in that it proposes that every device or application contain a type system to describe the structure and format of the data that can be communicated. This internal application type system is combined with features which allow each device or application to perform dynamic type negotiation with external applications, files, or systems. The XPL type system is self-referencing, which allows it to capture formal application protocols without requiring a large amount of supporting documentation or references to many other specifications. Flexibility has not been compromised to ensure that protocols based on the REST architectural style or RPC mechanisms can be developed.
1.1. Components of the proposal

The two major components of the proposal are as follows:

1. XPL, a powerful Extensible Presentation Language which is used both at protocol design time and by applications at runtime via very simple and small code libraries to communicate both primitives and compound types. XPL creates an internally consistent directed graph, which may be cyclic, to define complete versioned type systems to be stored with (or notionally with) an application or device.

2. XPL/TRP, a compact Type Resolution Protocol, in turn provides (i) dynamic negotiation of protocols and their constituent types between disparate devices; (ii) protocol versioning all the way down to the type level; and (iii) dynamic discovery of device capabilities and versions. An XPL device is able to describe and communicate its own application protocol. All these features are built-in to the protocol and intrinsic to its operation, rather than being extra-cost additions to it.

In XPL and XPL/TRP, after a short handshake in which data representations and common versions of types are negotiated between communicating peers, data is transmitted directly; completely free of metadata; and at the latest common version that can be negotiated, at every level of type.

1.2. Benefits

The benefits of the techniques described herein are many:

1. Beyond the initial handshake, transmission space overheads are zero and data exchange is space-optimal.

2. Time overheads are minimal and are confined to the first exchange between any two devices, or when type version levels change on either side.

3. Space overheads (implementation code space and bootstrap data space) are minimal (a few kilobytes: compared to the far larger requirements of even a SAX parser).

4. API libraries have been designed and implemented in support of these claims, which also demonstrate that application programming can also be made very simple. XPL scales to around 6kb of code and data in current implementations, and can be scaled to less than 1kb of code and data for even smaller devices.
5. Duty cycle (and hence power) requirements are therefore minimal.

6. Devices can be interrogated for their capabilities, expressed as the types they understand, via protocol primitives already built into the system for other purposes.

7. Conversations take place over the most recent version possible of every compound type: trivially, the message itself; and, recursively, every compound type of which it is composed, down to the level of primitives.

8. Being formally defined, XPL can readily be processed automatically, by dictionary compilers/integrity checkers; to generate type-specific marshalling and unmarshalling code; or to generate gateway code, e.g. to and from schema-based XML representations of the same data.

9. XPL and XPL-TRP can also contain discovery of atomic type formats such as word order, floating-point format, etc., and negotiation of a common format for communicating each native type. This feature allows the ‘Internet of Things’ to contain heterogeneous devices without reliance on ‘network byte order’, fixed-width integers, specific presentation rules for floating-point, and so on.

1.3. Potted History

The fundamental aspect of all information exchange and protocol design is agreement on how the information to be exchanged is encoded. Many methods have been developed to design and implement new application protocols:

1. In early solutions this agreement was handled by documented specifications that were then implemented by hand. This is still the case today for many file formats and other information exchange. Any change to a specification can cause ripple effects through multiple applications.

2. ASN.1 or Abstract Syntax Notation provides an abstract description (or contract) of the information to be exchanged. This is then encoded using a number of different encodings. There are also ASN.1 tools which use the ASN.1 data to generate source code to read and write the encoded information. Formal specification systems like this provide a much better solution to ensuring multiple parties can create software that is interoperable.
3. CORBA was intended to be the final solution for building distributed systems; it would create a network of applications that could work together. However, CORBA can only describe services using the formal Interface Definition Language (IDL). Information external to CORBA required a different design and strategy. Using CORBA tools, the IDL specifications can be used to generate software to allow clients to access services. The combination of IDL+IOP completely specifies the wire format of CORBA messages.

4. XML and XML Schema were born out of the Standard Generalized Markup Language. The W3C standards organisation defines XML as "a simple, very flexible text format derived from SGML (ISO 8879)." XML’s main feature of being an easily readable text based format is its biggest advantage and also in some respects its biggest disadvantage.

5. Simple Object Access Protocol (SOAP) was developed as a way to leverage XML for application communication. It brought the advantage of complex data that was not available with CORBA, and the ability to work within the web framework. It has recently been criticised for becoming too large with many different related specifications under the WS-* moniker. It is well-supported in several major programming languages, albeit with rather heavyweight tools.

6. RESTful Web Services is one of the latest in a long line of solutions to finding a common method of specifying application-level communication. The REST architectural style embodied in HTTP provides a number of basic designs which provide protocol developers a framework to work within. However, REST web services need to be combined with structured data and other documentation to provide a complete system. It requires more hand-written software than is required with systems like CORBA and SOAP to implement a complete application.

Any application protocol which might become a standard solution for an industry must take into consideration the totality of the solution. This is especially the case for constrained "Internet of Things" devices. All levels of the protocol must be clearly defined including: transport, security, session, presentation and application protocols. However, in addition to the base protocol many semantic details must be captured in documentation. Solutions like SOAP and CORBA are able to reduce the documentation as the higher levels of the protocol are captured in the system. However, REST style systems generally require more documentation to complete an application protocol. This additional documentation also translates to more development time required for each new protocol devised. Any data
which is transferred between peers at any layer above the transport protocol forms part of the application protocol and should be considered as part of the complete system. Finally, the code size must also be taken into consideration.

The issue of versioning is common to all of these methods of information exchange. Changes to the file format, ASN1 definition, CORBA interface, or XML schema can create high costs in maintaining and updating a distributed computing system. This is because the information about the format of the information is stored external to the application. The metadata or description of the protocol is lost in the translation from protocol description to implementation.

1.4. XPL Introduction

XPL was created to combine (i) the extensibility provided by XML and XML Schema, (ii) the ability to describe and encode binary information like IDL+IOP and ASN.1, (iii) the ability to create remote procedure call services like CORBA, and (iv) to be inherently ‘version-aware’, to ensure that a change to one aspect of the system did not create cost across the whole network.

The XPL solution is to have each application in the network contain metadata about all the information it can communicate (send and receive). This allows each XPL application to negotiate directly with other peers the information they are able to exchange. It is this fundamental change of moving the metadata knowledge into the application or file, rather than keeping it externally, which is the most important aspect of XPL. By embedding the metadata (knowledge about communications) into applications we have created a system which mimics human communication. Like two people who speak multiple languages meeting for the first time and negotiating which language to speak, XPL allows two computers to negotiate directly what data they can transfer. This creates a highly flexible system that is open to change.

The ability to have embedded metadata in each application or peer has many advantages for industry associations developing new application protocols. When device certification is required, the device to be certified can be quickly queried for all of its capabilities. Optional functionality can be easily determined and tested. Companies are also able to introduce additional functionality to create market differentiation while still complying with specific industry standards.

XPL versions each datatype individually to ensure maximum flexibility when change is required. It becomes trivial to amend a protocol by adding a new version of a subtype rather than having to version the
entire protocol, with all the attendant time and standardization overheads that implies.

XPL uses binary data. By providing metadata that describes the fundamental concepts of binary encoding we are able to create a general, flexible, and extensible system.

XPL only deals with the syntax and metadata associated with encoding information. It does not specify the constructs required to handle service oriented architectures or the transport of the encoded information. This allows XPL to be used in any method of information exchange, such as peer to peer communications, file formats, and message queue systems. The subtype-oriented architecture provides flexibility and re-use across protocols to maintain small code and data space for any XPL node.

At the core of XPL is a set of 35 metadata statements which are used to define each other, via some mutual recursion. This forms a directed cyclic type graph with no external definitions and creates a sort of Rosetta Stone from which all other information can be defined. These core 35 metadata statements constitute a 'meta dictionary'. It is this concept which allows two peers to discover from first principles whether and how the information they wish to communicate matches. The core 35 types may also be extended to create new ways to define the format of information.

Each device in an XPL network contains the core set of 35 metadata statement and metadata describing any other data it can communicate. The structure of the XPL type system allows for multiple versions of any data type to be defined and for the extension of complex data types. This gives each device knowledge of all of the information types it can communicate.

The XPL type system embedded (or notionally embedded) in each device is combined with methods for performing type comparisons with external systems. The type comparisons are designed to find the common definition type set shared by two XPL type systems. The shared definitions are maintained in type maps between the internal type system to the external. In this document, two methods have been proposed to build type maps. The XPL Type Resolution Protocol (XPL/TRP) provides dynamic resolution for communicating devices, while the XPL Message Format (XPL/MF) allows a file to specify the XPL type system of the information encoded. The XPL/MF is not a requirement for the "Internet of Things"; however, it is included in the XPL specification for completeness. Additional methods of mapping the XPL type system could also be created to map to XML; however this is not defined in this proposal. XPL may also be used to describe non-dynamic protocol descriptions; as is the case of using XPL to
describe the XPL Type Resolution Protocol (XPL/TRP) in this document. XPL can be used for building both service protocols (e.g. discovery) and application protocols of any design. This includes but is not limited to request/response, publish/subscribe and one way event based messages. Without entering into the REST/SOAP debate, XPL may also be used to design REST-based application protocols which could be knitted into the fabric of the web.

XPL is applicable to all communications, from the smallest embedded devices to large message oriented corporate systems. The aim is to create a world that every device and application can describe how it is able to communicate with other applications. We have proven that an XPL enabled device can shrink to as small as 6kb, or grow to service-oriented architectures. XPL is specifically designed for the ‘Internet of Things’, i.e. the concept that every electronically device will eventually mesh across our homes and businesses and create a global network of interconnected devices.

2. Basic Concepts

Any protocol can be considered as a union of types, where the types are the constituent messages of the protocol. Those message types themselves have subtypes, and so on recursively down to the atomic types, when closure occurs. In other words, a protocol is itself a type which is in turn comprised of other types. In general, the whole constitutes not a tree but a directed graph of types (as subtypes are generally re-used). In full generality, this graph is not necessarily acyclic. A fully general type mechanism of this kind needs to be able to cope with directed type graphs both acyclic and cyclic.

Any protocol is subject to versioning as new requirements or old problems are identified. A fully general versioning type processor would apply versioning to the constituent subtypes all the way down to the atomic types.

The implications of this are profound:

1. Not only can an outermost type (a protocol) be versioned; so can all its constituent sub-protocols and subtypes, down to the primitive types.

2. New features can be introduced into old protocols simply by the addition of a new version of the required subtype, or by addition of further concrete types to the abstract type mapping of an existing type. This very significantly reduces the costs of protocol versioning in time, space, and money.
3. It also implies that peers can communicate in any mix of an old and new protocol that satisfies version compatibility at all the sublevels.

To establish type version agreement, two communicating versioning type processors have to traverse their type trees to identify the most recent version of each type that the processors have in common. This traversal must be bottom-up so as to start with leaf types and finish with the uppermost type in the tree, which is the protocol itself. To handle cyclic type graphs, a backwards and forwards reference mechanism is also required such that (i) types can refer backwards to already-negotiated types, and (ii) types can create forward references to types yet to be resolved, in the case where both types concerned form part of a cycle of types.

Therefore the type resolution & versioning algorithm must (i) create all necessary forward type references, (ii) traverse the type tree bottom-up, and (iii) resolve forward references. In addition, the type resolution must be capable of uniquely identifying each node uniquely in the directed graph of types.

The XPL type system provides facilities for grouping or types, uniquely naming and versioning of types and recording the definition of simple and compound types. Although this document doesn’t specify algorithms, it is noted that a reverse topological sort produces the correct ordering in the absence of type cycles.

2.1. Type Library Locations

The requirement of performing generalised tree traversal and comparison between two type trees places special requirements on the structure of the type tree. Specifically, each type definition must be (a) uniquely addressable, and (b) clearly separate the definition so that it may be compared with remote types. XPL accomplishes this requirement using location/definition pair entries for every entry in the type library. In some cases a "composeable" type may be made of multiple entries in the type library.

The location of a type in the directed type graph must uniquely address each data type in the versioned type library. To accomplish this, the type library uses four different location specifiers. These are:

1. dictionary.base - Specifies the base of the type library. There must only be a single entry that specifies this location in any one type library.
2. **dictionary.name** - Specifies a non-versioned identity in the type library. A non-versioned type entry is a naming group similar to Java package names.

3. **dictionary.definition** - A versioned data type that includes a type name and version. Each type has a major and minor version identifier.

4. **dictionary.relation** - Some compound data types are composeable. They are composed of a single dictionary.definition and additional dictionary.relation types.

Combined, these four location specifiers are able to uniquely address and separate each definition in the type library. This is essential to allow type agreement methods such as those used in XPL/TRP and XPL/MF to form type agreements on subsections of the type library.

### 2.2. Type Maps

A second requirement for performing generalised tree traversal and comparison between two type trees efficiently, involves type identifiers. Each entry in a device type library has an associated identifier. However, as each device may have different definitions in their type library, the associated identifier will also be different. As it would be impossible to uniquely assign all data type identifiers from a central type bureau another method is required. A "Type Map" provides a method to map internal type entry identifiers to external agreed type entry identifiers.

The "Type Map" concept is specifically required when using the XPL/TRP or XPL/MF method of establishing type agreement. It allows through dynamic assignment a loose coupling and tight binding to be created between devices. Using either the XPL/TRP or XPL/MF method, type identifiers are mapped between internal and externally agreed type identifiers. To illustrate the "Type Map" concept two partial type libraries are presented:

<table>
<thead>
<tr>
<th>id</th>
<th>type name</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>uint32</td>
<td>unsigned 32-bit integer</td>
</tr>
<tr>
<td>34</td>
<td>book</td>
<td>book compound data type</td>
</tr>
<tr>
<td>35</td>
<td>author</td>
<td>book author string</td>
</tr>
</tbody>
</table>

**Table 1: Device 1 (Partial Type Library)**

This simplified type library includes three type identifiers with
corresponding type names. The type definition is a simple description and is not how XPL formally defines data types. A second device has the following:

<table>
<thead>
<tr>
<th>id</th>
<th>type name</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>uint32</td>
<td>unsigned 32-bit integer</td>
</tr>
<tr>
<td>42</td>
<td>book</td>
<td>book compound data type</td>
</tr>
<tr>
<td>44</td>
<td>author</td>
<td>book author string</td>
</tr>
</tbody>
</table>

Table 2: Device 2 (Partial Type Library)

Using a "Type Map" these two systems can create a common identifier that may be used in communication to uniquely identify the data type.

<table>
<thead>
<tr>
<th>Device 1 id/type</th>
<th>Agreed Identifier</th>
<th>Device 2 id/type</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 - uint32</td>
<td>21</td>
<td>41 - uint32</td>
</tr>
<tr>
<td>34 - book</td>
<td>22</td>
<td>42 - book</td>
</tr>
<tr>
<td>35 - author</td>
<td>23</td>
<td>44 - author</td>
</tr>
</tbody>
</table>

Table 3: Device 1 & 2 Type Map

To assign an agreed identifier the peers must find type agreement and ensure that the entries are equivalent. Equivalence is established by comparing the type entry location and definition information. If unique type entry locations are not found to be equivalent, no agreed identifier can be assigned. Similarly, if the type entry locations are found to be the same, yet the definitions are different, no agreed identifier can be assigned. In effect each device creates their own Type Map as such:

<table>
<thead>
<tr>
<th>Device 1 Type Map</th>
<th>Device 2 Type Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 - 33:21</td>
<td>21:41 - uint32</td>
</tr>
<tr>
<td>author - 35:23</td>
<td>23:44 - author</td>
</tr>
</tbody>
</table>

Table 4: Device Type Maps

When dealing with resource constrained devices the agreed identifier may be dictated by the server. This reduces complexity on small
devices and allows the protocol to be treated as a static definition. It is up to the client to match the definitions and identifiers assigned by the resource constrained device. If "Device 2" in the case above was a resource constrained device the type map would be:

```
<table>
<thead>
<tr>
<th>Device 1 Type Map</th>
<th>Device 2 Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 - 33:41</td>
<td>41 - uint32</td>
</tr>
<tr>
<td>book - 34:42</td>
<td>42 - book</td>
</tr>
<tr>
<td>author - 35:44</td>
<td>44 - author</td>
</tr>
</tbody>
</table>
```

Table 5: Constrained Device Type Maps

A Type Map is not only used to perform dynamic agreement between applications. The Type Map may be used to describe static protocols. In these situations the data types used by the protocol are pre-arranged and contain no dynamic aspects. The base XPL Type Resolution Protocol is an example of this type of protocol which requires a static pre-determined Type Map.

3. Extensible Presentation Language

XPL describes the format of binary data using metadata defined by the XPL type system. It is sufficiently powerful and flexible to be applied to network protocols, file formats, or indeed the format of any information.

It may be extended at various points to allow the description of new data formats not currently supported.

3.1. Metadata

An XPL Type Library is a collection of metadata used in reading and writing binary data. The Type Library is composed of a set of entries. Each entry in a library contains a unique identifier, a unique name or location, and a data structure representing its metadata definition. This is embedded with an application or device to provide detailed metadata descriptions for sending and receiving information. XPL uses a binary encoding system for data and metadata to allow machines to communicate faster and easier. XML type libraries can be maintained via a native editor, or, as in existing implementations, via a textual programming syntax and a compiler. However, when developing XPL dictionaries a text-based description is convenient to describe and compare definitions.
3.1.1. XPL Text Format Grammar

The XPL text grammar included and used throughout this document provides the most basic requirements needed for the purpose of this document. Additional facilities would be required for practical use but are not discussed herein. The grammar is shown below.

expression: (" IDENTIFIER (primary)* ")

primary: expression
   | '[' ( primary)* ']
   | '#' IDENTIFIER
   | IDENTIFIER ':' value

value: INT
   | QSTRING

IDENTIFIER: ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'.'|'_')*
INT: ('0'..'9')*
QSTRING: '"' ( .)* '"'

3.1.2. Primitive Types

Primitive types have the general form of:

<type>:<value>

Examples of this include:

    uint16:23    meta.name:"remote.rpc.request"  u8utf8:"text"

A type may be a compound data type and be read using the primitive type format. This is the case for both "meta.name" type and "u8utf8" type shown above. These types will be described in more detail later.

3.1.3. Compound Types

Compound datatypes are those that are made of multiple elements.

All compound types have the general form:

    (<type name> <param 1> ... <param N>)

An instance of a compound type is:

    (dictionary.definition meta.name:"somename" meta.version:"1.0")
This instance is a "dictionary.definition" type with two parameters. The first parameter is the type "meta.name" with value "somename". The second parameter is the type "meta.version" with the value "1.0".

3.1.4. Array Types

Arrays require repeating elements. The general form of an array is:

[ <element 1> <element 2> ... <element N> ]

An example instance of an array of meta.name type would be:

[ meta.name:"test" meta.name:"foo" ]

This shows an array of two "meta.name" elements with the value "test" and "foo" respectively.

3.1.5. Reference Identifiers

There are many situations where a reference to another type identifier is required. A reference to a type identifier has the form:

#<type>

An instance of a reference is:

#uint8

The #uint8 is replaced with the mapped identifier for the selected type. Reference identifiers will be explained more fully later.

3.2. Type Library Structure

The structure of the dictionary plays an important role in supporting the functionality provided by XPL. Every entry in the dictionary is made up of a unique location and a definition. This clear separation between location and definition ensures that the definition metadata used for marshalling is clearly separated from metadata required by the library.

The type library structure is a directed graph with each location in the graph being uniquely addressable. The base graph forms a tree structure; the following provides a small example tree structure based on a simple set of types to describe a book.
base
   +----- book (cluster)
      +------ id
          |   +----- id (version 1.0)
          |
      +------ catno
          |   +----- catno (version 1.0)
          +------ book_detail
                 +----- book_detail (version 1.0)

This simplified structure shows a "base" node to identify the base of the type library, a "book" cluster (or package) and three types: "book.id", "book.catno" and "book.book_detail". Each of the types have a single Version 1.0 entry. The definition of each type is not included for simplicity.

3.2.1. Type Library Entries

As described earlier, a type library is made up of entries. Each entry is made up of a location and definition and is assigned a unique identifier. As the assignment of the identifier is dynamic only the location information and definition is defined. The following is the general form for all library entries:

(library.entry
     <location>
     <definition>)

3.2.2. Type Library Locations

There are a number of different ways to specify a unique location in the type library. Every location specified in a dictionary must be unique and contain a definition.

3.2.2.1. library.base

The library.base type is a special location; it specifies the base of a library. There must be only a single entry in the type system with this location. The library.base definition should be a meta.cluster (ie package) to allow other types to be added. The meta dictionary first data type is defined as such.

(library.entry
     (library.base)
     (meta.cluster))
3.2.2.2. library.name

The library.name location specifies a uniquely named type. A library.name location is used to define definitions that are not version controlled. The meta.cluster definition is one type which is not version controlled.

(library.entry
  (library.name meta.name:"book")
  (meta.cluster))

3.2.2.3. library.definition

Any datatype that requires version control is added to the meta dictionary with a library.definition location. The library.definition datatype includes a meta.name and meta.version. The "book.catno" type might be defined as:

(library.entry
  (library.definition meta.name:"book.catno" meta.version:"1.0")
  (meta.reference #u8utf8))

The meta.reference to u8utf8 states that the type is a short UTF-8 string. Definitions will be described later.

3.2.2.4. library.relation

A relation location is used when defining composeable types. It allows additional metadata to be attached to an already defined versioned definition. The target definition must be designed to accept relation types for this type to be allowed; the meta.abstract is an example of such a type. The following shows an example of attaching a "meta.abstract_map" definition to a previously defined "book.id".

(library.entry
  (library.relation
    meta.name:"book.id"
    meta.version:"1.0"
    tag:"supplierid")
  (meta.abstract_map #book.supplierid))

This meaning of this will be covered during the discussion of abstract data types.
3.3. XPL Metadata Definitions

3.3.1. Primitive Types

XPL defines the following primitive types in the common type library.

- **bool**: a Boolean represented in 8 bits: 0 represents false; 1 represents true
- **uint8**: unsigned 8-bit integer
- **int8**: signed 8-bit integer
- **uint16**: unsigned big-endian 16-bit integer
- **int16**: signed big-endian 16-bit integer
- **uint32**: unsigned big-endian 32-bit integer
- **int32**: signed big-endian 32-bit integer
- **uint64**: unsigned big-endian 64-bit integer
- **int64**: signed big-endian 64-bit integer
- **float**: IEEE-754 signed floating point number (32 bits)
- **double**: IEEE-754 signed double precision floating point number (64 bits)
- **u8utf8**: an array of UTF-8-encoded characters of maximum length 255 bytes.

XPL defines primitive types using the meta.atom metadata type. For example, the uint16 data type is defined as:

```
(library.entry
  (library.definition meta.name:"uint16" meta.version:"1.3")
  (meta.atom
    (uvint28:16 /* min bit-length */
    (uvint28:16 /* max bit-length */
    [ (meta.attribute.fixed_length uvint28:16)
      (meta.attribute.integer)
      (meta.attribute.unsigned)
      (meta.attribute.bigendian)])

The meta.atom type includes a minimum and maximum bit length followed by an array of attributes which describe the data type. In this case
the uint16 data type is an Integer, is Unsigned, is Bigendian and has a fixed length of 16 bits. Different attributes may be used or created to define different primitive types. XPL describes the binary encoding of data types, so the uint16 data type is encoded directly using its binary format. For example, the value 166 is encoded in binary as:

```
0 0 0 0   0 0 0 1   0 1 1 0   1 1 1 0
```

XPL uses twos-complement representation for signed types, and big-endian or ‘network’ byte ordering by default. Implementations may also support ones-complement, little-endian or other data representations through the extension of the meta library attribute types.

3.3.2. References

The most fundamental expression used in XPL is the reference. A meta.reference is simply a reference or pointer to a data type that is already defined in the type library. The following demonstrates the definition of meta.id which defines a reference to a uvint28 data type.

```plaintext
(library.entry
  (library.definition
    meta.name:"meta.id"
    meta.version:"1.3")
  (meta.reference #uvint28))
```

3.3.3. Compound Types

XPL supports compound types in a similar way to XML-Schema: sequences, arrays, and choices, of primitive types and/or already defined compound types. This is powerful enough to define general data structures.

Sequence, array, and choice being built-in types, they are defined at the ‘meta’ level, as meta.sequence, #meta.array, and meta.abstract respectively. Formal definitions will appear later.

3.3.4. Arrays

A meta.array consists of the array length followed by the array elements. The type-arguments of a meta.array are an expression for the length and an expression for the type:
In this case, the above describes a type "value_list" version 1.0 that is an array of uint16 using a length expression of uint8. To illustrate, here is an instance of the above array:

```
[ uint16:255 uint16:16 uint16:8 ]
```

When encoded to a stream this would look as follows (in hexadecimal notation):

```
0x03 0x00 0xFF 0x00 0xF0 0x00 0x08
```

The first byte (0x03) specifies the length, while the following bytes specify the data.

### 3.3.5. Sequence

A sequence is a compound type used to describe a sequence of elements. A sequence may contain references to types or other expressions. For example:

```
(library.entry
 (library.definition
   (meta.name:"sequence_value"
    meta.version:"1.0")
 (meta.sequence [
   (meta.reference #uint16)
   (meta.array
    (meta.reference #uint8)
    (meta.reference #uint16))]))
```

In this case the type "sequence_value" version 1.0 includes a uint16 value followed by an array. An instance of this type is:

```
(sequence_value uint16:8 [ uint16:12 uint16:255 ] )
```

A sequence is encoded with the data encoded directly with no additional data between elements. The above instance would be encoded to stream as follows (in hexadecimal notation):

```
0x00 0x08 0x02 0x00 0x0C 0x00 0xFF
```
The first type bytes contain the value 8, the third byte is the length of the array and the final four bytes are the two values of the array.

3.3.6. Tag

The meta.tag type is used to provide descriptive information to elements in sequences or other parts of a structure. It does not change the underlying data encoding. It can be used for documentation or other purposes, for example to assist the binding of elements to a specific language. For instance, a Java marshaller would match the meta.tag descriptions with a Java bean’s getter and setter names. Consider the following definition of ‘mytype’:

(library.entry
 (library.definition meta.name:"mytype" meta.version:"1.0")
 (meta.sequence [
   (meta.reference #u8 )
   (meta.reference #s8 )
   (meta.reference #u16 )]))

An obvious problem here is that each element in the sequence is not named. This can be fixed using meta.tag:

(library.entry
 (library.definition meta.name:"mytype" meta.version:"1.0")
 (meta.sequence [
   (meta.tag u8ascii:"dayOfMonth" (meta.reference #u8) )
   (meta.tag u8ascii:"changeFactor" (meta.reference #s8) )
   (meta.tag u8ascii:"age" (meta.reference #u16) )]))

(The long-form of all the definitions in this section includes meta.tags for every item, as well as surrounding library declarations.)

3.3.7. Encoding

A meta.encoding is an ‘encoding’ annotation, e.g. ASCII, UTF-8, ISO-8859. In the example below a UTF-8 string type is defined. The meta.encoding type simply assigns an encoding name to an expression.

(library.entry
 (library.definition meta.name:"u8utf8" meta.version:"1.3")
 (meta.encoding
   (meta.array
    (meta.reference #uint8)
    (meta.reference #uint8))
   encoding:"UTF-8"))
The meta.array expression defines an array of uint8 bytes with a length defined by a uint8. The meta.encoding type does not modify the encoding of the array; it provides additional information to the application of how to process the data.

3.3.8. Envelope

The meta.envelope type is used to wrap a data type in an array. This allows the data type to be read without decoding it. This feature is required to support proxies, among other uses.

    (library.entry
     (library.definition
      meta.name:"data_envelope"
      meta.version:"1.3")
     (meta.envelope
      (meta.reference #uint16)
      (meta.reference #meta.definition)))

The meta.envelope type has two parameters. The first expression specifies the size of the array. The second parameter specifies the type of data encoded in the array.

3.3.9. Abstract

The meta.abstract data type provides a type of choice mechanism. It is a composeable data type, meaning that it can be made up from multiple entries in the type library. Composeable types provide a mechanism to extend the definition of a type without modification to a base definition.

The meta.abstract type is an important concept in XPL. It allows a type to be defined which has no associated definition. A meta.abstract_map entry is used to associate a concrete representation with an abstract data type. This allows XPL to be expanded by the user without modifying types previously defined. Concrete types may be mapped as part of the abstract definition or added later in separate definitions.

The meta.abstract data type is encoded using the concrete identifier of the data type followed by the value of the data type. This is an example of a type which requires the interaction with a dynamic type mapping mechanism such as XPL/TRP or XPL/MF. It may only be used in protocols which define static mappings for abstract type identifiers.

This use and encoding of meta.abstract types are best explained by an example. Suppose we have types book.catno and book.isbn already defined: we could define an abstract book.id type which maps to
either book.catno or book.isbn, as follows:

(library.entry
  (library.definition meta.name:"book.id" meta.version:"1.0")
  (meta.abstract [
    (meta.abstract_map #book.isbn)
    (meta.abstract_map #book.catno)]))

Another type can then reference the "book.id" type, for example a "book_list" array might be defined as:

(library.entry
  (library.definition meta.name:"book_list" meta.version:"1.0")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #book.id)))

An instance of this list might look like (assuming both book.isbn and book.catno are strings (u8utf8)):

[ book.isbn:"123123" book.catno:"1234-6789" ]

This instance of book_list would be then encoded as hexadecimal:

0x02 0x10 0x06 "123123" 0x11 0x09 "1234-6789"

In this case the first byte specifies the length of the array. The second byte specifies the type "book.isbn", the third byte is the length of the book.isbn string. Following the first string, the "book.catno" type identifier is encoded. This is followed by the length and value of the "book.catno" string. In this instance it is assumed that the 0x10 is the mapped external identifier of the "book.isbn" type and 0x11 is the mapped external identifier of the "book.catno" type.

Additionally or alternatively, further mappings can be added to 'bookid' later. For instance, a new type "book.supplierid" could be mapped to bookid via the 'library.relation' tag:

(library.entry
  (library.relation
    meta.name:"bookid"
    meta.version:"1.0"
    u8utf8:"supplierid")
  (meta.abstract_map #book.supplierid))

Composeable types are a powerful method of creating flexible data type libraries. It allows previously defined data types to stay the
same while extending a type to new functionality. This is important as it allows a protocol description to be extended to new functionality while keeping compatibility with systems that are unaware of the new changes.

3.4. Meta Dictionary Description

The meta dictionary is an important concept in XPL. It defines the subset of data types used to describe other data types as well as describe its own data types. This is best illustrated with examples of meta dictionary entries. The following defines the definition for meta.array:

(library.entry
  (library.definition
    meta.name:"meta.array"
    meta.version:"1.3")
  (meta.sequence 
    (meta.tag u8utf8:"size" (meta.reference #meta.expression))
    (meta.tag u8utf8:"data" (meta.reference #meta.expression))))

In this case the "meta.array" type is a sequence of type elements. The first is the size and the second is the data. The definition for meta.sequence is as follows:

(library.entry
  (library.definition
    meta.name:"meta.sequence"
    meta.version:"1.3")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #meta.expression))

The meta.sequence type is defined using the meta.array type. It is simply an array of up to 255 meta.expression types. These definitions are cyclic and require that both a defined by a type library via a bootstrap.

The meta.expression argument we have seen in meta.sequence and meta.array is a placeholder for types:

    meta.expression ::= meta.reference | meta.tag | meta.sequence |
    meta.array | meta.envelope | meta.encoding

It can best be understood initially in the same way as the 'expression' non-terminal in a grammar, hence its name. The meta.expression type is defined using meta.abstract as follows.
As an abstract type it has been mapped to all of the built-in methods of describing structure. As abstract types may be extended it also allows the structure of the type library to be extended.

The meta.identified type is an example expression that extends the meta dictionary. It can be used to specify that any valid datatype can follow. This is useful for elements which can be any datatype, rather like a CORBA ‘Any’. It is defined using two entries; the first defines the data type while the second adds the mapping to the meta.expression data type:

```
(library.entry
  (library.definition
    meta.name:"meta.identified"
    meta.version:"1.3")
  (meta.sequence [ 
    (meta.tag u8ascii:"description" (meta.reference #u8utf8))]))
```

Another abstract type in the meta dictionary is the meta.definition type. The specific concrete types are mapped to the abstract type. The abstract type meta.expression is also mapped, allowing any concrete type mapped to that type to be used as a meta.definition.
3.5. Library Types

The library types are used to make it easier for developers to define datatypes without being concerned with some of the complexities of the dictionary.location and dictionary.entry datatypes defined by the meta dictionary. These library types have been used previously to define type entries. However, types from the "library" type cluster are different to how type library entries are recorded and transferred during communication. The following describes how these types are mapped into the meta dictionary types.

NOTE: Many library types may be able to be removed or deprecated. They were more useful in a previous version of the meta dictionary.

3.5.1. library.base

The library.base datatype is a replica of the dictionary.base datatype. It is here for consistency in naming.

(library.entry
  (library.definition u8utf8:"library.base" u8utf8:"1.3")
  (meta.sequence []))

3.5.2. library.name

The library.name datatype is a replica of the dictionary.name datatype.

(library.entry
  (library.definition
    meta.name:"library.name"
    meta.version:"1.3")
  (meta.sequence []")

      This is also a location that the meta dictionary may be extended with new definition data types.

This is also a location that the meta dictionary may be extended with new definition data types.
3.5.3. library.definition

The library.definition datatype is a replica of the dictionary.definition datatype.

(library.entry
  (library.definition
    meta.name:"library.definition" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"name" (meta.reference #meta.name))
    (meta.tag u8utf8:"version" (meta.reference #meta.version))
  ])
)

3.5.4. library.relation

The library.relation is mapped to a dictionary.relation type. The id and version is mapped to the correct identifier for the relation.

(library.entry
  (library.definition
    meta.name:"library.definition" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
    (meta.tag u8utf8:"version" (meta.reference #meta.version))
    (meta.tag u8utf8:"tag", (meta.reference #u8utf8))
  ])
)

3.5.5. library.entry

The library.entry is mapped to a dictionary.entry. Using a dictionary entry removes the need for the developer to specify the identifier for each type or use an envelope around the definition.

(library.entry
  (library.definition u8ascii:"library.entry" u8ascii:"1.3")
  (meta.sequence [
    (meta.tag u8ascii:"location"
      (meta.reference #library.location))
    (meta.tag u8ascii:"definition"
      (meta.reference #meta.definition))
  ])
)
3.5.6. library.list

The library.list datatype is a basic list structure to put all datatype definitions. It is used as an intermediate step before converting all entries into a dictionary file.

(library.entry
 (library.definition u8ascii:"library.list")
 (meta.sequence [
   (meta.array
     (meta.reference #uvint28)
     (meta.reference #library.entry))]))

3.6. Meta Dictionary Reference

The meta dictionary defines the core set of data type definitions used in XPL. Each type must only use meta data which is also defined by the meta dictionary.

3.6.1. library.base

The library.base entry defines the base meta.cluster of the library. Every library must contain a single unique library.base entry.

(library.entry
 (library.base)
 (meta.cluster))

3.6.2. uint8

The uint8 definition defines an unsigned eight bit integer.

(library.entry
 (library.definition meta.name:"uint8" meta.version:"1.3")
 (meta.atom uvint28:8 uvint28:8
   [ (meta.attribute.size uvint28:8)
     (meta.attribute.integer)
     (meta.attribute.unsigned)
     (meta.attribute.bigendian) ] ))

3.6.3. uvarint28

The uvint28 type defines an unsigned 28 bit integer. Variable length encoding is used with a minimum of one octet and a maximum of four octets. The MSB of each byte is used as a continuation bit. When the MSB of a byte is set to 1 it indicates that the following octet is also part of the integer.
The meta cluster definition defines the base group of which all meta data types are members.

The meta.id type is used to specify all XPL datatype identifiers. It uses the uvint28 datatype for encoding.

The meta.cluster datatype is used to define a cluster of entries in the dictionary.

The meta.abstract_map type is used to define mappings from abstract datatypes to concrete representations. This may be used by either the meta.abstract datatype or independently by specifying a dictionary.relation type location.

An abstract type may be mapped to other abstract types. The target abstract type must inherit all sibling mappings. Circular references MUST NOT be allowed.
3.6.8.  meta.abstract

The meta.abstract type specifies that a type is abstract and has multiple concrete types. An abstract type is encoded to a byte stream by first encoding the meta.id of the concrete type. The type identifier is followed by the encoding of the mapped datatype. The definition may include an initial set of mappings to concrete or other abstract datatypes. Additional types may be mapped to an abstract type using a dictionary.relation location.

3.6.9.  u8utf8

The u8utf8 type is a utf8 string of up to 255 octets long. The size of the string is encoded before the length using a uint8 type. The length specifies the number of raw octets used by the encoding, not the number of characters encoded.

3.6.10.  meta.name

Each datatype name is encoded using a meta.name type. A name must specify a cluster identifier that the type belongs. The name is a utf8 string. White space and "." must not be used in the name.
3.6.11. meta.version

Each datatype version is encoded using a meta.version type. A datatype has a major and minor version.

3.6.12. meta.definition

Each datatype is defined using a meta.definition type. The meta.definition type is abstract with mappings to meta.atomic, meta.abstract, meta.abstract_map and all types mapped to the abstract type, meta.expression. Additional concrete definitions may be added using the dictionary.relation location.

3.6.13. meta.expression

The meta.expression abstract datatype is used to define elements in the schema. It is mapped to meta.reference, meta.tag, meta.sequence, meta.array, meta.envelope and meta.encoding. Additional expression types may be added using the dictionary.relation location and meta.abstract_map definition.
3.6.14. meta.reference

The meta.reference datatype encodes the identifier of a selected type.

3.6.15. meta.tag

The meta.tag type is used to add names to sub-elements in a sequence or other part of a type definition. It does not modify the actual encoding of the data structure in any way.

3.6.16. meta.sequence

The meta.sequence type is used to describe a group of elements in a sequence. The encoding uses no meta information between elements.
3.6.17. meta.array

The meta.array type is used to describe any repeating array structure. The size is an expression which returns the number of elements to be read. The data specifies the expression of the repeating element.

(library.entry
 (library.definition
  meta.name:"meta.array" meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"size" (meta.reference #meta.expression))
   (meta.tag u8utf8:"data" (meta.reference #meta.expression))]))

3.6.18. meta.envelope

An envelope is used to wrap data structures. The size specifies an expression for the size of the envelope. The type specifies the datatype contained in the envelope.

(library.entry
 (library.definition
  meta.name:"meta.envelope" meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"size" (meta.reference #meta.expression))
   (meta.tag u8utf8:"type" (meta.reference #meta.expression)) ]))

3.6.19. meta.encoding

The meta.encoding type specifies the name of the encoding used in a particular data expression. This can be used to label an octet array as ASCII or UTF8 or other encoding.
3.6.20. meta.atom

Atomic datatypes are those that can not be broken up any further. An atomic datatype has a minimum and maximum bit length and a group of attributes. The atomic metadata is not usually used as part of the implementation; however it should provide an implementer with enough information to write marshalling code.

3.6.21. meta.atomic_attribute

The meta.atomic_attribute is an abstract datatype to describe any attributes of an atomic type.
3.6.22. meta.attribute

The meta.attribute cluster.

(library.entry
  (library.name meta.name:"meta.attribute" )
  (meta.cluster))

3.6.23. meta.attribute.size

(library.entry
  (library.definition
    meta.name:"meta.attribute.size" meta.version:"1.3")
  (meta.sequence [ 
    (meta.tag u8utf8:"size" (meta.reference #uvint28))
  ]))

3.6.24. meta.attribute.integer

(library.entry
  (library.definition
    meta.name:"meta.attribute.integer" meta.version:"1.3")
  (meta.sequence []))

3.6.25. meta.attribute.unsigned

(library.entry
  (library.definition
    meta.name:"meta.attribute.unsigned" meta.version:"1.3")
  (meta.sequence []))

3.6.26. meta.attribute.bigendian

(library.entry
  (library.definition
    meta.name:"meta.attribute.bigendian" meta.version:"1.3")
  (meta.sequence[]))

3.6.27. dictionary

The dictionary cluster.

(library.entry
  (library.name meta.name:"dictionary")
  (meta.cluster))
3.6.28. dictionary.base

The dictionary.base type is an identifier for the base of the dictionary.

(library.entry
  (library.definition
    meta.name:"dictionary.base"
    meta.version:"1.3"
  )
)

3.6.29. dictionary.name

The dictionary.name type is used to specify a simple name location in the dictionary. A dictionary.name location must only specify a meta.cluster definition in a dictionary. Internally, a dictionary.name is used as the identity of any dictionary.definition.

(library.entry
  (library.definition
    meta.name:"dictionary.name"
    meta.version:"1.3"
  )
)

3.6.30. dictionary.definition

The dictionary.definition type specifies a versioned data type definition.

(library.entry
  (library.definition
    meta.name:"dictionary.definition" meta.version:"1.3"
  )
)

3.6.31. dictionary.relation

The dictionary.relation type is used to relate definition information to already defined datatypes. The "id" is used to specify the type that the definition relates. The "tag" must be a utf8 string which is unique to the definition.

NOTE: Strictly speaking this is not required by the meta dictionary as this type is not used. However, it is required to attach other
datatypes to the dictionary and if not available in the meta dictionary would need to be defined separately as a meta dictionary extension.

(library.entry
  (library.definition
    meta.name:"dictionary.relation" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
    (meta.tag u8utf8:"tag" (meta.reference #u8utf8))
  ]))

3.6.32. dictionary.location

All entries in the meta dictionary must have a unique location. The dictionary.location type is abstract and defines four initial location types. The dictionary.location may be extended to use other location types if required.

(library.entry
  (library.definition
    meta.name:"dictionary.location" meta.version:"1.3")
  (meta.abstract [
    (meta.abstract_map #dictionary.base)
    (meta.abstract_map #dictionary.name)
    (meta.abstract_map #dictionary.definition)
    (meta.abstract_map #dictionary.relation)
  ]))

3.6.33. meta.definition_envelope

The meta.definition_envelope type is used in the dictionary to wrap all definition information. This is required so that inter-dependent types locations may be read before attempting to read datatypes.

(library.entry
  (library.definition
    meta.name:"dictionary.definition_envelope" meta.version:"1.3")
  (meta.envelope
    (meta.reference #uvint28)
    (meta.reference #meta.definition)))

3.6.34. dictionary.entry

Each entry in the dictionary is recorded as a dictionary.entry. This type records the id, dictionary.location, and dictionary.definition for each datatype.
3.6.35. dictionary.entry.list

The dictionary.entry.list is an array of dictionary entries.

3.7. Meta Dictionary Mappings

To allow systems to compare and build on the meta dictionary the meta dictionary datatypes must be mapped to identifiers. The following specifies the base meta dictionary typemap that must be used for any systems using datatype agreement methods.
The default mappings for meta dictionary types.

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>base</td>
</tr>
<tr>
<td>1</td>
<td>uint8</td>
</tr>
<tr>
<td>2</td>
<td>uvint28</td>
</tr>
<tr>
<td>3</td>
<td>meta</td>
</tr>
<tr>
<td>4</td>
<td>meta.id</td>
</tr>
<tr>
<td>5</td>
<td>meta.cluster</td>
</tr>
<tr>
<td>6</td>
<td>meta.abstract</td>
</tr>
<tr>
<td>7</td>
<td>meta.abstract_map</td>
</tr>
<tr>
<td>8</td>
<td>meta.u8utf8</td>
</tr>
<tr>
<td>9</td>
<td>meta.name</td>
</tr>
<tr>
<td>10</td>
<td>meta.version</td>
</tr>
<tr>
<td>11</td>
<td>meta.definition</td>
</tr>
<tr>
<td>12</td>
<td>meta.expression</td>
</tr>
<tr>
<td>13</td>
<td>meta.reference</td>
</tr>
<tr>
<td>14</td>
<td>meta.tag</td>
</tr>
<tr>
<td>15</td>
<td>meta.sequence</td>
</tr>
<tr>
<td>16</td>
<td>meta.array</td>
</tr>
<tr>
<td>17</td>
<td>meta.envelope</td>
</tr>
<tr>
<td>18</td>
<td>meta_encoded</td>
</tr>
<tr>
<td>19</td>
<td>meta.atom</td>
</tr>
<tr>
<td>20</td>
<td>meta.atom_attribute</td>
</tr>
<tr>
<td>21</td>
<td>meta.attribute</td>
</tr>
<tr>
<td>22</td>
<td>meta.attribute.size</td>
</tr>
<tr>
<td>23</td>
<td>meta.attribute.integer</td>
</tr>
<tr>
<td>24</td>
<td>meta.attribute.unsigned</td>
</tr>
<tr>
<td>25</td>
<td>meta.attribute.bigendian</td>
</tr>
<tr>
<td>26</td>
<td>dictionary</td>
</tr>
<tr>
<td>27</td>
<td>dictionary.base</td>
</tr>
<tr>
<td>28</td>
<td>dictionary.name</td>
</tr>
<tr>
<td>29</td>
<td>dictionary.definition</td>
</tr>
<tr>
<td>30</td>
<td>dictionary.relation</td>
</tr>
<tr>
<td>31</td>
<td>dictionary.location</td>
</tr>
<tr>
<td>32</td>
<td>dictionary.definition_envelope</td>
</tr>
<tr>
<td>33</td>
<td>dictionary.entry</td>
</tr>
<tr>
<td>34</td>
<td>dictionary.entry_list</td>
</tr>
</tbody>
</table>

Table 6: Meta Dictionary Mapping Table
3.8. Meta Dictionary Encoding

The following shows the meta dictionary encoded to a dictionary.entry_list using the mappings defined above. The encoding is a mixture of hex and ASCII. Where a single character is displayed the value is in ASCII (eg. u i n t 8 ); where two values are displayed next to each other the value is in hex (eg. 23 00 1b 01 05).

To conform to this specification, an implementation MUST encode the meta dictionary exactly as printed here when encoding the meta dictionary.

23 00 1b 01 05 01 1d 00 05 u i n t 8 01 03 09 13 08 04 16 08 17 18 19 02 0d 07 u v i n t 2 8 01 03 09 13 08 1c 04 16 08 17 18 19 03 lc 00 04 m e t a 01 05 04 1d 03 02 i d 01 03 02 0d 02 05 1d 03 07 c l u s t e r 01 03 02 0f 00 06 1d 03 0c a b s t r a c t _ m a p 01 03 08 0f 01 0e 02 i d 0d 04 07 1d 03 08 a b s t r a c t c l u s t e r 01 03 07 0f 01 10 0d 01 0d 06 08 1d 00 06 08 u t f 8 01 03 0c 12 10 0d 01 0d 01 05 U T F 2d 8 09 1d 03 04 n a m e 01 03 13 0f 02 0e 05 g r o u p 0d 04 0e 04 n a m e 0d 08 0a 1d 03 07 v e r s i o n 01 03 14 0f 02 0e 05 m a j o r 0d 01 0e 05 m i n o r 0d 01 0b 1d 03 0a d e f i n i t i o n 01 03 07 05 05 03 13 07 06 0c 1d 03 0a e x p r e s s i o n 01 03 08 07 06 0d 0e 0f 10 11 12 0d 1d 03 09 r e f e r e n c e 01 03 04 0f 01 0d 04 0e 1d 03 07 t a g 01 03 12 0f 02 0e 04 n a m e 0d 08 0e 04 d a t a 0d 0c 0f 1d 03 08 s e q u e n c e 01 03 07 0f 01 10 0d 01 0d 0c 10 1d 03 05 a r r a y 01 03 12 0f 02 0e 04 s i z e 0d 0c 0e 04 t y p e 0d 0c 11 1d 03 08 e n v e l o p e 01 03 12 0f 02 0e 04 s i z e 0d 0c 0e 04 t y p e 0d 0c 12 1d 03 08 e n c o d i n g 01 03 16 0f 02 0e 04 d a t a 0d 0c 0e 08 e n c o d i n g 0d 08 13 1d 03 04 a t o m 01 03 7 0f 03 0e 0e m i n _ b i t _ l e n g t h 0d 02 0e 0e m a x _ b i t _ l e n g t h 0d 02 0e 00 a t t r i b u t e s 10 0d 01 0d 14 14 1d 03 0e a t o m _ a t t r i b u t e 01 03 06 07 04 16 17 18 19 15 1c 03 09 a t t r i b u t e 01 05 16 1d 15 04 s i z e 01 03 0a 0f 01 0e 04 s i z e 0d 02 17 1d 15 07 i n t e g e r 01 03 02 0f 00 18 1d 15 08 u n s i g n e d 01 03 02 0f 00 19 1d 15 09 b i g e n d i a n 01 03 02 0f 00 1a 0c 00 0d a t e 01 05 1b 1d 1a 04 b a s e 01 03 02 0f 00 1c 1d 1a 04 n a m e 01 03 0a 0f 01 0e 04 n a m e 0d 09 1d 1d 1a 0a d e f i n i t i o n 01 03 15 0f 02 0e 04 n a m e 0d 09 0e 09 07 v e r s i o n 0d 0a 1e 1d 1a 08 r e l a t i o n 01 03 0f 0f 02 0e 02 i d 0d 04 0e 03 t a g 0d 08 1f 1d 1a 08 l o c a t i o n 01 03 06 07 04 1b 1c 1d 1e 20 1d 1a 13 d e f i n i t i o n _ e n v e l o p e 01 03 05 11 0d 02 0d 0b 21 1d 1a 05 e n t r y 01 03 1e 0f 03 0e 02 i d 0d 02 0e 04 n a m e 0d 1f 0e 0a d e f i n i t i o n 0d 20 22 1d 1a 0a e
4. XPL Type Resolution Protocol

XPL/TRP uses the same type agreement protocol to communicate between client and server as it does using message files and dictionaries. In both cases a Type Map is built to map internal identifiers with external type identifiers used during communication. However, in the case of XPL/TRP network agreement the Type Map is agreed dynamically. To perform these agreements the XPL Type Resolution Protocol is used. The protocol has a small number of request/response pairs which allow any client and server to agree on the data structures they plan on transferring to each other. The XPL/TRP can be transmitted on a completely separate channel to the actual data transfer channel, and in cases of extreme need can be carried out offline, e.g. by implanting flash memory.

The XPL/TRP Network Protocol has a very simple structure. It uses a single uint8 byte to identify a message type, followed by the actual message data.

4.1. Message Types

The following message types are defined:

- Check Core
- Map Type
- Map Default
- Map Reserve
- Map Reverse
- Error
- Message

Each of these is described further in the following sections. It is important to note that the current XPL/TRP uses ‘server-dictated’ data type version selection. This means that the version of a data type used on the communication link is dictated by the server.

1. It will be seen that, between them, the Map Type, Map Default, Map Reserve, and Map Reverse operations together provide a completely general mechanism to resolve type versioning in
directed type graphs, specifically including cyclic type graphs.

2. Although not defined here for clarity of exposition, it is almost certainly necessary for efficiency reasons to define ‘multi-type’ versions of these three operations, such that instead of sending a single type to be mapped/reserved/reversed as above, an array of types is sent, and the response contains an array of mapping-ids. This is a simple extension to the operations shown above. The only necessary rules are (i) the type array must be ordered in the same way as it would be if sent one at a time using the operations above, and (ii) all Multi Map Reserve and Multi Map Reverse operations must precede all Multi Map Type operations for types that contain subtypes needing to be reserved or reversed.

3. It is further envisaged that instead of providing a new mapping-id or an error for an unknown type, a specific implementation may choose to provide a mapping to the type meta.opaque. This technique is useful where a participating device isn’t interested in the data in the type concerned, but wants to be able to receive types (message) that contain it. This technique would provide a way (i) for the sending device to send the enclosing type, and (ii) for the receiving device to receive that data, skip over it, and ignore it, while being able to receive the rest of the message.

4.2. Protocol Definition

The version field listed must have the static value 0x10 (major/minor version 1.0).

4.2.1. Request

The XPL/TRP is defined by the following:
4.2.2. Response

(library.entry
 (library.definition meta.name:"trp.response" meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"version" (meta.reference #uint8))
   (meta.choice [
     (meta.choice_option uint8:0x01
      (meta.reference #trp.check_core_dictionary_response))
     (meta.choice_option uint8:0x02
      (meta.reference #trp.check_core_version_response))
     (meta.choice_option uint8:0x03
      (meta.reference #trp.map_response))
     (meta.choice_option uint8:0x04
      (meta.reference #trp.map_default_response))
     (meta.choice_option uint8:0x05
      (meta.reference #trp.map_reserve_response))
     (meta.choice_option uint8:0x06
      (meta.reference #trp.map_reverse_response))
     (meta.choice_option uint8:0x07
      (meta.reference #trp.map_error))
     (meta.choice_option uint8:0x08
      (meta.reference #trp.message))])])
)
4.3. Check Core

An XML/TRP client MUST send this as its first message to an XML/TRP server. It contains the binary data for the core meta dictionary. The server is able to use this to confirm that both client and server are using the same version of XPL. This also establishes the base data types required to agree on other data types.

This message checks if the core meta dictionary data types are the same. These data types are assigned a common set of identifiers that both client and server must adhere to. This operates as a bootstrap mechanism for other types to be defined. This bootstrap mechanism allows the core metadata to be expanded, i.e. to include new concepts for how to describe the data being transferred that were not previously present in the core metadata.

The client issues a check core request. The response should either be the encoded meta dictionary for the client to check or an indication of the version of meta dictionary used by the server.

In very small devices the response could be a URL which references the data dictionary used by the device.

4.3.1. Request

(library.entry
 (library.definition
   meta.name:"trp.check_core_request" meta.version:"1.3")
 (meta.sequence []))

4.3.2. Response

the response MUST be either a trd.check_core_dictionary_response or trd.check_core_version_response.

A client receiving a trd.check_core_dictionary_response MUST perform a binary compare on a local copy of its meta dictionary. If the core meta dictionary type map does not match, the client and server cannot continue. There is no basis for on-going data agreement. The communications channel MUST be closed.
If a client receives a `trd.check_core_version_response`, the client MUST check it contains the same meta dictionary version. If the client does not contain the same version there is no basis for ongoing data agreement. The communications channel MUST be closed.

4.4. Map Type

When a client needs to send a datatype that is not in the meta dictionary list of types for the first time, it MUST send a "Map Type" request message to the server. This request contains the type’s location and structure. The server receives the message, finds the datatype and decides if the data structures match. If they match, an identifier is assigned for the channel for that type; if they don’t match, an error MUST be returned to the client. If an agreement cannot be found, then the client will receive an error and the message being sent MUST be aborted.
4.4.1. Request

(library.entry
 (library.definition
   meta.name:"trp.map_request"
   meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"location"
    (meta.envelope
     (meta.reference #uvint28)
     (meta.reference #dictionary.location)))
   (meta.tag u8utf8:"definition"
    (meta.envelope
     (meta.reference #uving28)
     (meta.reference #dictionary.definition))))

4.4.2. Response

The server responds with the mapped identifier:

(library.entry
 (library.definition
   meta.name:"trp.map_response"
   meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"id"
    (meta.reference #meta.id))])

If the meta definition uses data structures that have themselves not been mapped, these MUST be mapped before a request can be completed. This process MUST occur in a bottom-up fashion, i.e. a Map Type request MUST NOT refer to a type which consists of types which have not yet been mapped. (Note that it is possible to resolve circular chains of references to unmapped types, via the Map Reserve request described below.)

4.5. Map Default

A single conversation must only use a single version of any one data type. The map default request is used to resolve which version of a selected type is used. The dictionary.location must be a dictionary.base, dictionary.name or other location specifier without version information.

4.5.1. Request

A request consists of a dictionary location wrapped in a meta.envelope.
4.5.2. Response

The response consists of the mapped identifier, dictionary.definition location and dictionary.definition. The client MUST confirm that the returned location and definition match its local type library.

4.6. Reserve Type

The Map reserve request is used for the special case that a type data structure’s definition is self-referencing or creates a dependency loop within its definition. In these cases the client requires an identifier for a data structure before it can finish writing a definition. In this case only the type name (dictionary.name) is sent to the server.
4.6.1. Request

(library.entry
 (library.definition
   meta.name:"trp.map_reserve_request"
   meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"location"
   (meta.envelope
   (meta.reference #uvint28)
   (meta.reference #dictionary.location)))]

The server MUST respond with the mapped identifier, or an Error Message if the type name doesn’t exist at the server. The client MUST also eventually follow up with a Map request to confirm the definition of the reserved type.

4.6.2. Response

(library.entry
 (library.definition
   meta.name:"meta.protocol.map_reserve_response"
   meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"id"
   (meta.reference #meta.id)]))

4.7. Map Reverse

In some cases the server will send a new datatype that the client has not yet mapped. In this situation the client MUST perform a reverse mapping. The client MUST send the server the mapped identifier given to it by the server. The client MUST NOT send a mapped identifier that didn’t originate with the server.

4.7.1. Request

(library.entry
 (library.definition
   meta.name:"trp.map_reverse_request"
   meta.version:"1.3")
 (meta.sequence [
   (meta.tag u8utf8:"id"
   (meta.reference #meta.id)]))

The server MUST respond with the name and definition, or an Error Message if the mapped_id is unknown.
4.7.2. Response

(library.entry
  (library.definition
    meta.name:"trd.map_reverse_response"
    meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"location"
      (meta.envelope
        (meta.reference #uvint28)
        (meta.reference #dictionary.location))
    (meta.tag u8utf8:"definition"
      (meta.envelope
        (meta.reference #uving28)
        (meta.reference #dictionary.definition))))])

4.8. Message

The send message request is to allow using the same base protocol stack for other requests. This allows a single socket endpoint or other transport to be used. A request and reply simply include an envelope with an identified message type.

4.8.1. Request/Response

(library.entry
  (library.definition meta.name:"trp.message" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"message"
      (meta.envelope
        (meta.reference #uvint28)
        (meta.reference #meta.identified)))]

4.9. Error

Any request that results in an error on the server returns an error message. Error Message are presently implemented without error details (uint8:0x05), but the definition proposed here incorporate a 16-bit error code and a message string, as follows:
4.9.1. Response

(library.entry
  (library.definition
    meta.name: "trp.map_reverse_response"
    meta.version: "1.3")
  (meta.sequence [
    (meta.tag u8utf8: "errorCode" (meta.reference #uint16))
    (meta.tag u8utf8: "errorMsg" (meta.reference #u8utf8))]))

5. XPL Message Format

Argot message files are binary encoded files that provide the specification of their data with the data. An Argot file contains three parts; a meta dictionary, a data dictionary and the data. Argot dictionary files use this format to store dictionary definitions.

The Argot Message format allows the full specification of the data to be transferred with the data. This requires no external definition of the data. For an application to be able to read the file its type library must contain all the data types used in the file. A Type Map is generated from the data dictionary portion of the file to read the data.

The process of reading a file involves:

1. Binary compare of meta dictionary map. The very first dictionary map of the meta dictionary is the core meta dictionary. The only way to read this entry is by performing a binary compare. These are the base dictionary items used to describe new items. Please refer to the meta dictionary reference section for details of the core meta dictionary.

2. Build and read Meta dictionary. The core dictionary consists of an array of dictionary.entry_list values. After the first meta dictionary is read the additional dictionary.entry_lists area read and mapped. These include any additional types used to define types in the data dictionary.

3. Read the Data dictionary. Using the Type Map produced from entries in the Meta Dictionary the Data dictionary is read. A Data dictionary type map is created based on the types identified.

4. Read the Data. Using the Data dictionary type map the actual data of the file is read.
5.1. Message Format Type Description

{library.entry
  {library.definition meta.name:"dictionary.file" meta.version:"1.3"
    {meta.sequence [
      {meta.tag u8ascii:"core"
        {meta.array
          {meta.reference #uint8}
          {meta.reference #dictionary.entry_list}}
      {meta.tag u8ascii:"meta"
        {meta.array
          {meta.reference #uint8}
          {meta.reference #dictionary.entry_list}}
      {meta.tag u8ascii:"message"
        {meta.identified u8ascii:"message"}}
    ]}}

6. Acknowledgements

The authors wish to thank the ZigBee Alliance for the opportunity to present this work at the Melbourne workshop in October 2009 and for the valuable questions and comments from the floor.

The authors also wish to thank Tracii Kelly for editorial assistance.

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

This document is explicitly not concerned with security considerations and assumes that all communications take place via connections which once established exhibit privacy, integrity, authentication, and authorization to the extent necessary.

9. Informative References


[CORBA] Object Management Group, "CORBA: Common Object Request


Appendix A.  Additional Information

A.1.  Exclusions

This document is not concerned with: transport protocols; object addressing or protocol verbs (the REST vs SOAP debate); security; ...

The XPL-TRP requires a transport protocol which provides reliable delivery of messages. It can be used with UDP or TCP; however, the authors strongly recommend that in the case of UDP a separate protocol allowing guaranteed delivery be layered over it. While transport protocols are outside the scope of this proposal, the authors wish to acknowledge that 6lowpan networks may not be a suitable environment for a full TCP implementation. An alternative solution would be to provide a restricted TCP implementation with compressed headers in a similar fashion to that of 6lowpan IP header compression.

In addition, any security requirements are expected to be dealt with by a layer between the transport and XPL-TRP.
A.2. Usual Suspects

The 'usual suspects' SOAP, XML, and EXI are considered by the present authors to have very limited applicability as solutions to these issues, for the following reasons:

1. The implied requirement for SAX or DOM parsing, particularly the latter, imposes unacceptable costs. The costs are borne both in code space and data space in the devices concerned, as well as in time and therefore in duty-cycle and power consumption.

2. The data representation even under EXI is not space-optimal, which is considered a binding requirement in the application area concerned. Each XML or EXI message still contains its own metadata in the form of element and attribute names, and start/finish markers for containing elements. EXI must also be combined with other protocols and possibly extensive documentation for each protocol specification. EXI relies heavily on data compression for its space-efficiency. Impressive figures have been seen for reductions of XML messages to EXI, but of course their impressiveness relies entirely on the questionable practice of using XML and textual data in the first place. Further, they imply time overheads for encoding and decoding, and space overheads for the compression algorithm’s encoding trees.

3. Without the flexibility and meta-facilities afforded by DOM parsing and the Document Object Model, XML really lacks all interest as a candidate solution.

4. Versioning is only supported at the uppermost (schema) level.

5. Schemas are large compared to available memory, which is defined as being as little as 48-128k.

6. Schemas and documents rely on external entity declarations which have to be either dynamically fetchable or built into application resources. The entity declarations required for HTML presently add up to at least 31 kilobytes.

7. On the other hand, schemas are generally not built out of other schemas, so redundancy is high.

8. There is no inherent support for discovery of device capabilities and versions: this can be only added via an otherwise redundant application layer, or by probing to detect error conditions, i.e. conflating protocol error with protocol ignorance, which is epistemologically unsound.
9. By implication, these considerations also rule out SOAP as an element in the solution. The SOAP/WADL/WSDL approach with its implicit reliance on DOM and XML Schema is almost certainly complex to embed on very small devices.

10. A binary HTTP equivalent combined with Binary XML and perhaps some footprint-reduced form of SAX may possibly scale to small devices, although in our view this remains to be shown. In any case such a solution will certainly require additional layers of complexity to define and implement. Some form of WADL/WSDL and additional documentation will be required.

The present authors don’t really wish to participate in the endless REST/SOAP debate about how exactly endpoints, messages, RPCs, &c, should be expressed in and across public computer systems, but we suspect it is (a) possibly unresolvable and in any case (b) largely irrelevant to the present problem. Nor is it clear to us that space considerations will really permit applications in these devices to be object-oriented as far as the outside world is concerned, so the question of how to represent internal endpoints may not even arise.

However it should be stated here that XML/TRP is entirely capable of ‘peaceful co-existence’ with the REST viewpoint.

Authors’ Addresses

David Ryan (editor)
Live Media Pty. Ltd.
PO Box 4591
Melbourne, Victoria 3001
Australia

Email: david@livemedia.com.au

Esmond Pitt (editor)
Verismart Software Inc
351 California Street, Suite 400
San Francisco, CA 94104
United States of America

Email: esmond.pitt@verismartsoftware.com