Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

Ingeniería Informática. Especialidad sistemas distribuídos

PROYECTO FIN DE CARRERA

– DESIGN AND IMPLEMENTATION OF AN EXTENSIBLE BINARY ENCODING (XBE32) C LIBRARY –

Author: **LIA BAILAN ZAMORA**
Diplomada en Informática
Director: **MANUEL URUEÑA PASCUAL**
Licenciado en Informática

March, 2009

For those regarded as warriors, when engaged in combat, the vanquishing of thine enemy can be the warrior's only concern. Suppress all human emotion and compassion. Kill whoever stands in thy way, even if that be Lord God or Buddha himself. This truth lies at the heart of the art of combat.

Kill bill vol.1

Dedicated to Telmo, Liö-Liö and ...da boy.

Contents

List of Figures

Chapter 1

Introduction

Nowadays computer networks in the modern business industry are essential. There are a lot of services that in these days are provided through the network, and without them, it would be impossible to save resources by sharing them.

In addition to that, technology has advanced a lot, making possible for the users to possess machines (laptops, PC's, PDA's, etc) that are not static but can go with them wherever they go, forcing the IT research to discover new ways to provide the network services to those mobile devices. For all those reasons, service discovery protocols have been developed.

The access to network services scattered among different locations is possible thanks to Service Discovery technology. It is the technology responsible to find all the different resources attached to the user network and offer them to them in case are authorised to use them. Service Discovery Protocols are the protocols that carry out the mission. Some of these technologies are: Jini, Salutation, SLP, UPnP. This protocols have been conceived to allow the cooperation among devices/services with minimal human intervention.

To perform all the three tasks (plus the specific of each protocol) it is necessary the interchange of messages in the net between the different machines. To find the proper encoding mechanism for a Service Discovery Protocol is the main goal of this work.

1.1 Motivations

XSDF (Extensible Service Discovery Framework), published in 2005, defines an architecture with several entities and protocols for the management and location of Service information. XSDF intents to be the tool to give transparency to the different network operations that a user needs for its daily working. XSDF tries to address all these problems, offering a framework to find the best service for the user. That is, looking for the one which optimizes the needs of the user and of the network. In this way, this tool can be useful not just to find hidden resources, but to keep a load balance between all of them as well.
There is no need to say that, since XSDF is another SDP, the needs to implement it meets the needs of the others SDP. Therefore XSDF is forced to send encoded messages as mean of communication between the machines that take part in its process.

Nowadays, the most popular format to represent hierarchical structured information is the eXtensible Markup Language (XML), and it has been employed by multiple network protocols and applications. Although its textual representation allows protocols to be extensible, and eases development and debugging, it requires more bandwidth and processing than a binary counterpart.

On the other hand, we can mention ASN.1, at some extent less popular format, that gives us a representation that requires much less bandwidth, even saving space at the bit level, but has quite complex processing rules.

As a result, we have two very different encoding mechanisms that cover different user's necessities. XML provides a mean to represent hierarchical data; ASN.1 allows to encode the data in a minimum space so less bandwidth is needed. Our goal is the implementation of XBE32, which is meaning to cover the deficiencies of both of them, its purpose it is to give to the user the chance to represent hierarchical data in an smaller space.

The requirements asked to XBE32 will be:

- to be capable of expressing hierarchical structures in its messages.
- to save bandwidth through the creation of really compact messages.

This document specifies an eXtensible Binary Encoding (XBE32), a simple binary encoding for network protocols that carry hierarchical data. It pretends

to be an intermediate way between XML and ASN.1. In spite this encoding has already a Java implementation, we have seen necessary to make its correspondent implementation in C, due to the fact that this language is more extended in the Unix/Linux platform and above all, among the open source community, which in our humble opinion is a very important source for the development of this technology.

1.2 Objectives

The main objective of this work is to develop a library to encode/decode XBE32 messages in C language. This implementation will become a library that has to be simple, transparent and efficient. The library had to work as the base for any other network application that required to encode messages in XBE32.

Those were the official objectives that must cover the XBE32 implementation described in this work. For the student responsible to carry out this task, there are additional objectives. Among them the most important one was the improvement of the ability to program in C language, but there were others as the comprehension of a complex protocol and its posterior materialization, the application of the engineering methods learned during the degree courses, the use of distributed programming techniques, and the use and improvement of Linux O.S and several utilities as control version applications, scientific language processors, etc.

Finally, since the goals for this work have been explained, it is time to describe the means used to perform the implementation, just as the implementation itself has been described. In spite it has not been an easy task to conceive and to capture this library, the technology needed is not very sophisticated:

- A simple PC machine capable to host:
  - a gcc compiler.
  - a revision control system (subversion).
  - a simple text editor (gedit).
  - a latex compiler.
  - a graphic application capable of create UML diagrams (DIA).
  - an Internet connection (Just to do research).

1.3 Document contents

The structure of this document is the following:
In the present chapter, we present the problem we are trying to solve with this work, and the objectives that we want to achieve with the implementation. In addition to that, some background about the means we have at our disposal to perform the implementation is given.

After that, in the second chapter, we present some of the other encoding standards available in the market and we establish a comparison between them in order to show which are the points that are not covered by those alternatives that make our solution necessary.

The third chapter is the specification of the protocol implemented in this work. This chapter is included to give the users an overview of XBE32 and make them capable of evaluate the way the implementation has been done.

The fourth chapter offers an explanation about the implementation. Describes the system architecture and the functions with some graphical means and some examples to improve the reader understanding.

In the fifth chapter we summarize the initial goals and check if these have been achieved, look for the main difficulties during the job, and recount the future works opened by the realization of this implementation and the skills and techniques acquired with this work.

Finally, this writing counts with a couple of appendixes: Installation, which explains which files must be installed and how install and/or create them; and the man pages which constitute the users' manual.

Chapter 2

STATE 0F THE ART: ENCODING STANDARDS

2.1 The competitors

In the present time, XML is the most widely format to represent hierarchical information, and since this kind of information is the most popular in the current computer applications, XML has become the most extended format in the moment.

Development of XML started in 1996 and it has been a W3C Recommendation since February 1998, which may make us suspect that this is rather immature technology. In fact, the technology is not very new. Before XML there was SGML, developed in the early '80s, an ISO standard since 1986, and widely used for large documentation projects. The development of HTML started in 1990. The designers of XML simply took the best parts of SGML, guided by the experience with HTML, and produced something that is no less powerful than SGML, and vastly more regular and simple to use. In addition to this, XML allows protocols to be extensible, and eases development and debugging, but XML presents a great problem for distributed applications: it requires more bandwidth and processing than the expression of its information in/on binary data.

Over the past decade, XML has become the preferred encoding system of the major IT and business companies. It has received an enormous support from these, and this is the reason why it has become the most important encoding

language nowadays. It has been adopted by the great majority of Universities, and thus there is no single IT or Information science student who does not use it for at least a couple of projects. Its simplicity and its ease to deploy any hierarchical scheme have been its most valuable attributes to make from XML the standard encoding for any organization. On the other hand, we must remember that though XML is full of advantages, it is very difficult for just one technology to fulfil the goals of every user at every moment. XML consumes a lot of bandwidth and it is not suitable for small protocols which main goal is speed.

Since many users do not bother to check if one technology is the best for their applications (they take for granted that the most trendy always must be applied), some researchers have seen as something important to produce an study about XML and its suitability respect another old encoding standard, ASN.1. A fragment of this study [9], by Jose Angel Mart' inez Usero and Elsa Palacios Ramos, below in order to summarise the advantages and disadvantages of both standards:

> ASN.1 is designed to describe the structure and syntax of transmitted information content. ASN.1 provides the definition of the abstract syntax of a data element (or data type). The abstract syntax describes the syntactical structure and typed contents of data that are subsequently to be transmitted across some medium. The language is based firmly on the principles of type and value, with a type being a (non-empty) set of values. The type defines what values can subsequently be sent at runtime, and the value is what is actually conveyed across the medium at runtime.
>
> ASN.1 values are encoded before transmission using one of a number of different encoding mechanisms such as the Basic Encoding Rules (BER), the Distinguished Encoding Rules (DER), the Packed Encoding Rules (PER) [10] (ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)) or the recently introduced XML Encoding Rules (XER) [11](Information technology ASN.1 encoding rules:XML Encoding Rules). The encoding rules specify how the values of the abstract data types are converted into byte strings ready for transfer. The recipient must usually be aware of the type definition before receipt, as this is not transferred but must be inferred from the context in which the message exchange takes place.

The Basic Encoding Rules are very efficient and create Type, Length, Value (TLV) byte streams, so that the recipient, upon reading the length field, knows how many data bytes the value comprises. PER is even more efficient than BER, and is not based on TLV streams, so even greater optimisation can result. For example, PER never encodes the length of the value, unless it has to. If something has a fixed length, then the length field is not encoded.

During the transmission the ASN.1 data stream is never in a form readable by human operators (except when XER is used). Only when it has been transformed into some local data display format, prior to encoding or after decoding, can it be easily read by humans. In its behalf, it must be said that a lot of encoding rules can be used, as it has been said before. Among them, the XML encoding rules. But as a great inconvenience, we found that its binary encoding is rather complex and it must take several stages: ASN.1 encoding itself, and after that, the application of the selected encoding rules (PER, XER, BER or XML).

XML is a set of rules that allows data values to be encoded in text format. XML is a subset of the Standard Generalized Markup Language (SGML), but is also infinitely extensible. XML documents contain the information for transmission and consist of markup (which corresponds roughly to the "tag" and "length" parts in BER TLV encoding) and character data (which corresponds roughly to the "value" part in BER TLV encoding). Constraints can be imposed on the XML document structure with the provision of Document Type Definitions (DTD's) or XML Schemas. These describe the allowed markups that a conformant XML document can contain.

One can see immediately that XML is very verbose, and consequently creates large data streams. XML is transferred in textual format with no binary encodings or compression. Furthermore, the recipient has to examine every byte received in order to determine the end of a data value. However, XML goes through no transformations and remains in a constant human readable format throughout the process.

In some sense it can be said that DTD'os/schemas map to the abstract syntax type definitions within ASN.1 and the XML documents map to the ASN.1 encoded byte streams. There are a few major differences between ASN.1 and XML/DTDs, with XML/DTDs lacking

any concept of data type and ASN.1 being rich in built-in data types and supporting user-defined data types. Also, XML is very verbose, unlike ASN.1 encoding rules (except XER) that have been designed for optimal performance rather than human readability. However, from an application programmer'os perspective, XML is easier to debug since the data stream can be read without any special software tools. Trying to read an ASN.1 BER or PER byte stream is very complex, but a number of free tools do exist to display ASN.1 data in its original source form e.g. dumpasn1. In addition, the XML 1.0 specification is a lot newer, simpler and easier to understand than the ASN.1 documentation, which has gone through several iterations and therefore contains many more sophisticated features.

In many environments XML is a preferred way of encoding business transactions, since the messages are readily viewable by web browsers. If these environments involve simple XML messages, without digital signatures, then XML performs adequately and the benefits of XML can be realised. In fact we have found that simple XML message creation is more efficient than creating an equivalent ASN.1 byte stream. For critical real time systems where digital signing of complex data structures is required, and where performance is a key success factor, such as an electronic prescribing system for example, it has been shown that signed complex XML messages can be up to a 1000% slower to decode than an equivalent ASN.1 message.

XML is easy to manipulate and easy to understand, all factors which make it attractive to both senior management and developers. However, the key to many IT project failures has been the inability to perceive the needs of the end users, and performance is one of them. Some believe that in a real time system dealing in multiple transactions a second and requiring strong authentication through digital signatures, XML formatting is not a good protocol to choose. This might ultimately result in user dissatisfaction and perhaps even total system failure. Since end users are aware of system performance and not of the underlying data encoding mechanisms, we believe that performance figures are an important factor in system design. In several sources, it has been shown that with digitally signed messages ASN.1 can significantly outperform XML by over an order of magnitude.

2.2 Justifying the existence of XBE32

Despite XML is the most widely protocol used to encode data in Internet applications, as we have seen in the previous section, the performance of the mentioned protocol is not as good as it should be, at least in certain scenarios.

This is the main reason of the existence of XBE32. This protocol has been created in order to solve the problems raised by XML (bandwidth and processing).

Some of the characteristics that make XBE32 a better encoding system than XML are the following:

- XBE32 Elements are serialized inside TLV structures which are 32-bit aligned to ease the parsing process. As data is clearly delimited, XBE32 does not require to escape characters as XML does, thus it also facilitates message creation.

- XBE32 TLVs have a 2-octets long Type and Length fields. Therefore, XBE32 is well suited for simple protocols with short messages and a small set of identifiers. However, in order to be extensible, XBE32 Elements may also have variable-length names or longer binary identifiers. Moreover, XBE32 may support TLVs with an "unspecified" length in order to encode big messages, and to start sending a message before its total length is known. There are two kinds of XBE32 Elements: "Attribute Elements" which carry primitive data values, and "Complex Elements" which are not able to carry data by themselves but contain other Attributes and/or other Complex Elements.

- In order to be employed by modern programming languages, XBE32 make use of common primitive data types for its Attribute Elements, such as Strings, Booleans, Integers, Floats, as well as Arrays. Other data types can be encoded using the different binary Opaque value types defined by XBE32.

Once this point has been reached, an inspired reader could get to the conclusion that, if some recent studies have shown that ASN.1 could be the solution to the problems that XML raises, XBE32 has no reason to exist. Well, that is not that simple since there are several reasons that advises the use of XBE32

over ASN.1, the main one is simplicity, but we are going to enumerate others in the following list:

- ASN.1 data processing is further much complex than XBE32. Data is first encoded in the ASN.1 language, and after that, it is necessary to apply the mechanisms that converts the data following the PER, BER or XER encoding rules. XBE32 is an atomic encoding system that does not need of another mechanism or tools to encode the data, making the process quicker and easier to follow to the user.

- ASN.1 BER encoding is quite similar to XBE32 since it also employs TLVs. However ASN.1 BER is a lot more complex than XBE32 since its final objective is reducing bandwidth costs, not processing ones. For instance BER TLVs have variable length fields, and are byte-aligned. On the other hand XBE32 employs 32-bit aligned TLVs with fixed-length fields that greatly eases the parsing process. Moreover ASN.1 is a general encoding syntax not focused in a single domain of application but many. For instance it has 30 data-types and deals with globally unique OID identifiers, whereas XBE32 has been designed for simple protocols with a small set of identifiers and data-types.

Chapter 3

XBE32 SPECIFICATION

This chapter contains the full specification of the XBE32 encoding, the last version of the draft [1] written by Manuel Ureña Pascual and David Larrabeiti.

Nowadays, the most popular format to represent hierarchical structured information is the eXtensible Markup Language (XML), and it has been employed by multiple network protocols and applications. Although its textual representation allows protocols to be extensible and eases development and debugging, it could require more bandwidth and processing than a binary counterpart.

The eXtensible Binary Encoding (XBE32), a simple binary encoding for network protocols that carry hierarchical data. XBE32 Elements are serialized inside TLV structures which are 32-bit aligned to ease the parsing process. As data is clearly delimited, XBE32 does not require to escape characters as XML does, thus it also eases message creation. The final goal of this encoding is to be used by applications which need to reduce the bandwidth of the information sent and without great complications in the encoding process.

XBE32 TLVs have a 2-octets long Type and Length fields. Therefore, XBE32 is well suited for simple protocols with short messages and a small set of identifiers. However, in order to be extensible, XBE32 Elements may also have variable-length names or longer binary identifiers. Moreover, XBE32 may support TLVs with an "unspecified" length in order to encode big messages, and to start sending a message before its total length is known.

There are two kinds of XBE32 Elements: "Attribute Elements"

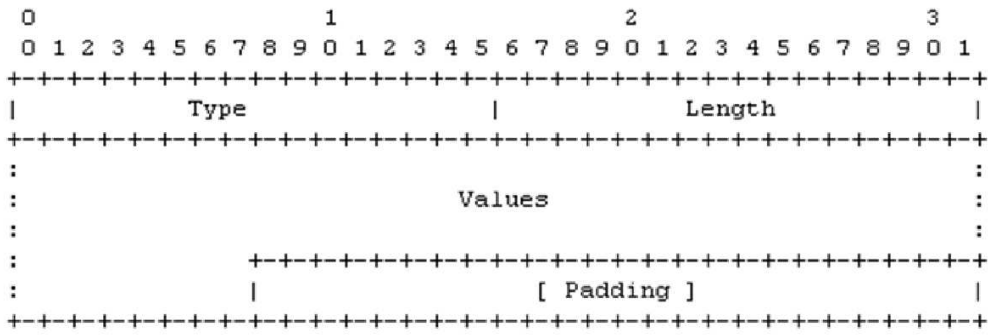

Figure 3.1: Format of a XBE32 TLV

which carry primitive data values, and "Complex Elements" which are not able to carry data by themselves but contain other Attributes and/or other Complex Elements.

In order to be used by modern programming languages, XBE32 employs common primitive data types for its Attribute Elements, such as Strings, Booleans, Integers, Floats, as well as Arrays. Other data types can be encoded using the different binary Opaque value types defined by XBE32.

The design of XBE32 has been conceived to intertwine two different layers. The first one will be responsible to handle the atomic parts of the language; the TLVs. The second layer will handle the elements; this elements are always composed by one or more of the latter.

Now, in the next sections we are going to introduce the different components of the protocol just in order to make more comprehensible the explanation about the way we have implemented it.

3.1 TLV format

A TLV (abbreviation of Type Length Value) is, as its name points out, a set composed by the type, length and value of the item we are trying to represent. Is the smallest item in the protocol, and the one from the others are made of. XBE32 Elements are encoded inside Type-Length-Value (TLV) structures, that MUST be aligned to 4-octet words. XBE32 TLVs could be: "Simple" TLVs if they carry primitive data values, or "Complex" ones if they contain other TLVs. The structure of a TLV is as figure 3.1 illustrates.

Type (16 bits):

This field describes the processing rules, TLV structure and what kind of data is carried inside the Values field. The Type field has the internal structure shown in figure 3.2:

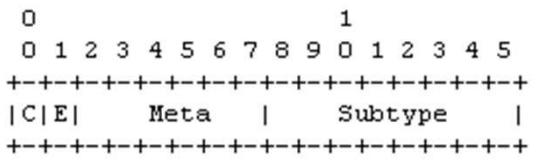

Figure 3.2: Dissected TLV type

C and E bits (1 bit each):

These two bits specify the measures that must be taken if the XBE32 processing entity does not recognize this Type value:

```
C - Continue Processing:
  0 - Discard this mandatory TLV and stop processing TLVs left
  1 - Skip this optional TLV and continue processing next TLV

E - Notify Error:
  0 - Do not report to the sender that this Type is unknown.
  1 - Report to the sender that this Type is unknown.
```

Meta (6 bits):

This subfield describes the internal structure of the TLV's Values field, as well as the type of the primitive data it contains 3.3:

Subtype (8 bits):

This subfield identifies the semantic meaning of this TLV and/or the data carried inside its Values field. Therefore, Subtype values should be defined by the upper application/protocol that is employing a XBE32 encoding. However, Subtype values 0x00 and 0xFF are reserved for XBE32 use and SHOULD NOT be employed for other purposes.

Length (16 bits):

This field MUST be encoded as an unsigned binary number in network byte order (a.k.a. Big Endian, i.e, the most significant byte first). It specifies the size in octets of the whole TLV structure, excluding padding. Length SHOULD be always equal or greater than 4 octets, that is, the length of the Type and Length fields. The

Type.Meta	TLV Values structure
0x00-0x1F	Multiple variable-length TLVs
0x20	Single variable-length opaque Value
0x21	Single variable-length string Value
0x24	Multiple opaque1 Values
0x25	Multiple int8 Values
0x26	Multiple boolean Values
0x28	Multiple opaque2 Values
0x29	Multiple int16 Values
0x2C	Multiple opaque4 Values
0x2D	Multiple int32 Values
0x2E	Multiple float32 Values
0x30	Multiple opaque8 Values
0x31	Multiple int64 Values
0x32	Multiple float64 Values
0x34	Multiple opaque12 Values
0x38	Multiple opaque16 Values

The unlisted values are reserved by XBE32 and SHOULD NOT be employed.

Figure 3.3: Primitive TLV types

only exception to this rule is a Complex TLV with a zero (0x0000) length value, whose meaning is explained in the next subsection.

Values and Padding (variable length):

The Values field may contain a single variable-length value, multiple fixed-length values, or other TLVs, as defined by the Type and Length fields. The Values field may be empty, that is, have zero octets. In that case, the Length field SHOULD be set to 4. In order to properly align a non-empty Values field to 4-octet words, up to 3 octets of padding space MUST be added and filled with zeros (0x00) in transmission, and they MUST be ignored in reception.

Unspecified Length

In some circumstances a message can not be delayed/stored and it must start being sent before all the data to be encoded is available. However, as a TLV header defines the total length of the structure, a TLV-encoded message should not be sent until all its data becomes available, or the total length can be inferred somehow.

For that reason, XBE32 parsers MAY allow a Complex TLV (i.e. containing other TLVs) to have an "unspecified" length. In that case, the last of the inner TLVs MUST be an End-of-data TLV to mark its ending. This "unspecified" length is indicated by setting the Length field of a Complex TLV to zero (0x0000).

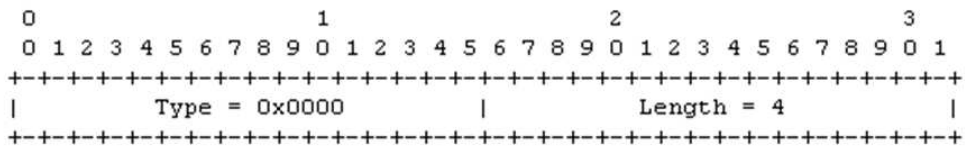

Figure 3.4: TLV signaling an End-of-data

End-of-data TLVs (figure above) have the Type field set to zero (0x0000) and a fixed Length of 4 octets, thus they MUST NOT include a Values field.

This optional mechanism allows XBE32 to encode Complex TLVs of arbitrary length. However, only Complex TLVs may have an "unspecified" length. The Values field of a Simple TLV containing primitive data Values MUST NOT be longer than 65532 octets.

3.2 XBE32 TLVs

Once we have defined a simple TLV, it is possible to describe the role that plays in the whole XBE32 encoding system. This section specifies all the possible TLV structures and data types allowed in XBE32. All TLVs share the common format for the Type and Length fields defined in the previous section, but the TLV showed before must be enriched to fulfill the needs of the encoding protocol. The main difference between XBE32 TLVs is the inner structure of their Values fields and the type of the primitive data they contain, as defined by the Meta part of the TLV's Type field.

Complex TLVs with inner TLVs

Figure 3.5 represents a Complex TLV containing multiple inner TLVs. If the Length is "unspecified" (i.e. zero), the Complex TLV MUST end with a 4-octet End-of-data TLV. Otherwise, if the Length of a Complex TLV is non-zero, it MUST NOT contain any End-of-data TLVs.

As XBE32 TLVs must be aligned to 4-octet words, all Complex TLV will be also aligned to 4-octet words. Therefore, padding MUST NOT be added, and the Length field SHOULD specify the size of the whole Complex TLV, including the length of all the inner TLVs it contains.

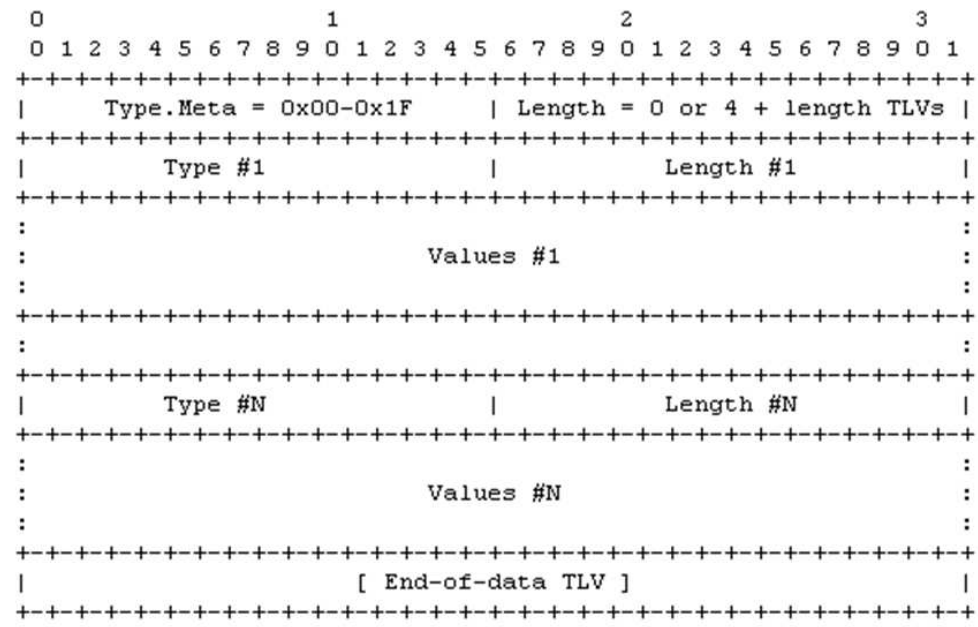

Figure 3.5: Complex TLVs with inner TLVs

However, as some of the inner TLVs Values may be padded, the Length of a Complex TLV SHOULD NOT be calculated as 4 plus the sum of the Length fields of all its inner TLVs, as these fields may not include their padding octets.

Simple TLV with one variable-length Value

Figure 3.6 represents a Simple TLV containing a single variable-length Value:

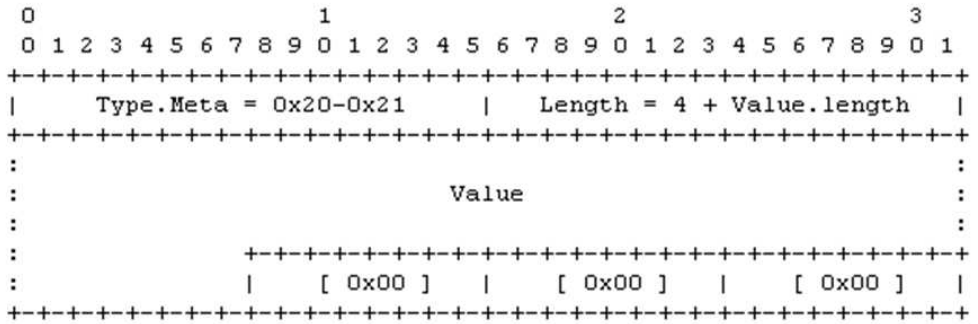

Figure 3.6: Example of simple TLV with one variable-length value

The Length field MUST specify the size of the Type and Length

fields, plus the length of the encoded Value measured in octets. If the Value is not aligned to 4-octet words, padding MUST be added. In that case, the Length field does not define the size of the whole TLV structure, but its total length without the padding octets

Simple TLVs with 1-Octet Values

Figure 3.7 represents a Simple TLV containing N, 1-octet Values:

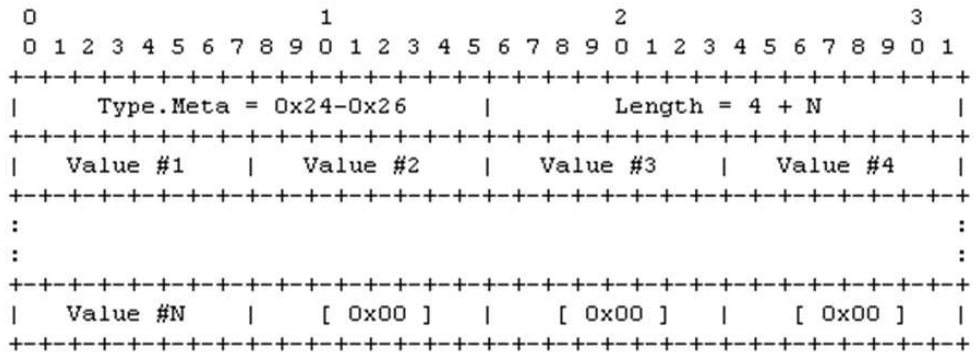

Figure 3.7: Example of simple TLVs with 1-Octet Values

The Length field MUST specify the size of the Type and Length fields, plus the number of 1-octet Values, if any. If the number of Values is not a multiple of 4, up to 3 padding octets MUST be added. In that case, the Length field does not define the size of the whole TLV structure, but its total length without the padding octets.

Simple TLV with 2-Octets Values

Figure 3.8 represents a Simple TLV containing N, 2-octets Values:

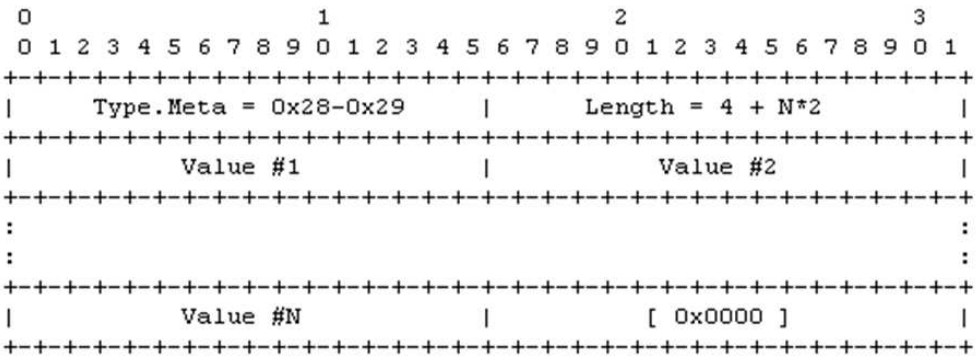

Figure 3.8: Example of simple TLV with 2-Octets Values

The Length field MUST specify the size of the Type and Length fields, plus the number of 2-octet values, if any, multiplied by two. If the number of Values is not a multiple of 2, two padding octets MUST be added. In that case, the Length field does not define the size of the whole TLV structure, but its total length without the padding octets.

Simple TLVs with 4-Octets Values

Figure 3.9 represents a Simple TLV containing N, 4-octets Values. As these TLVs are always aligned to 4-octet words, the Length field

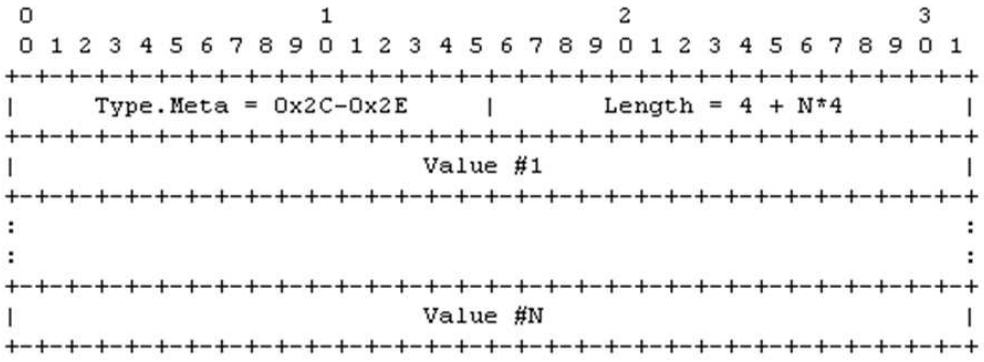

Figure 3.9: Example of simple TLV with 4-Octets Values

MUST specify the size of the whole TLV, and padding octets MUST NOT be added.

Simple TLVs with 8-Octets Values

As these TLVs are always aligned to 4-octet words, the Length field MUST specify the size of the whole TLV, and padding octets MUST NOT be added.

Figure 3.10 represents a Simple TLV containing N, 8-octets Values:

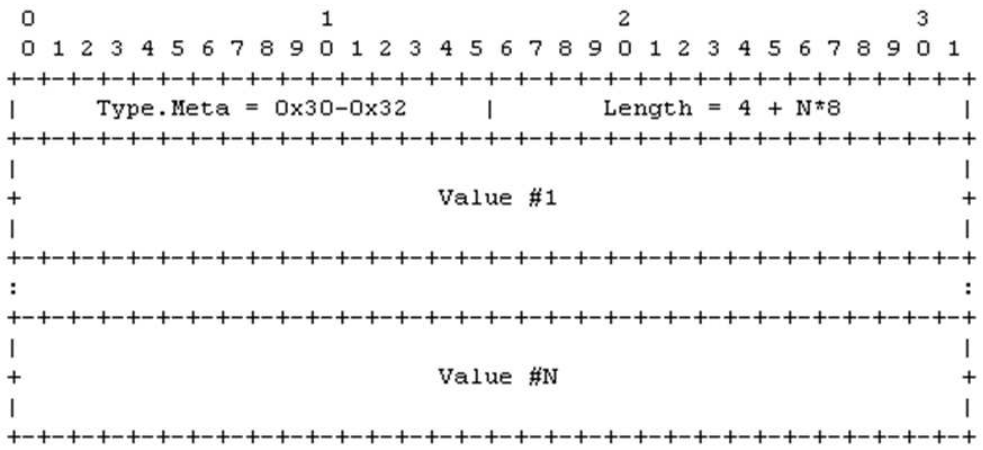

Figure 3.10: Example of simple TLV with 8-Octets Values

Simple TLVs with 12-Octets Values

Figure 3.11 represents a Simple TLV containing N, 12-octet Values:

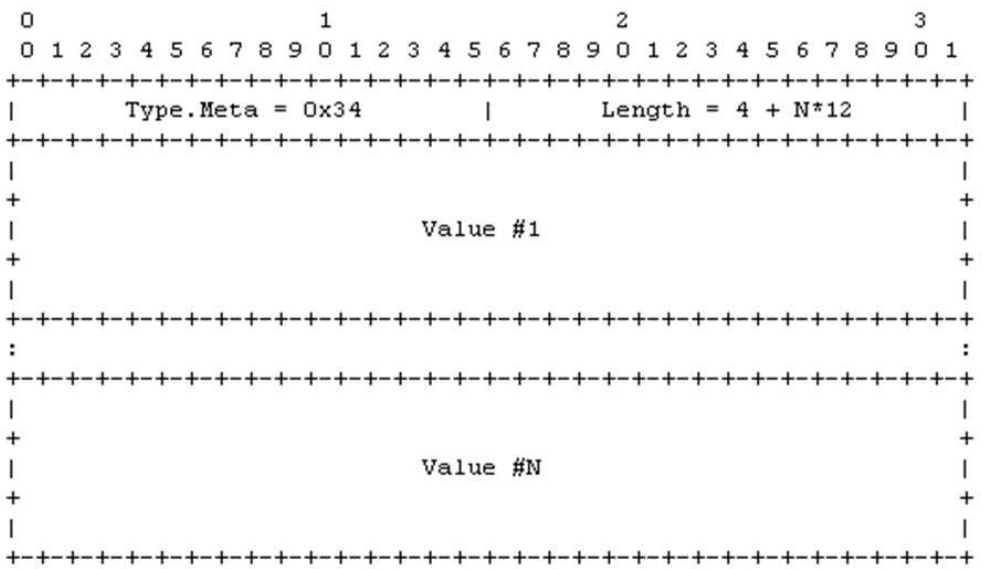

Figure 3.11: Simple TLVs with 12-Octets Values

As these TLVs are always aligned to 4-octet words, the Length field MUST specify the size of the whole TLV, and padding octets MUST NOT be added.

Simple TLVs with 16-Octets Values

Figure 3.12 represents a Simple TLV containing N, 16-octet Values:

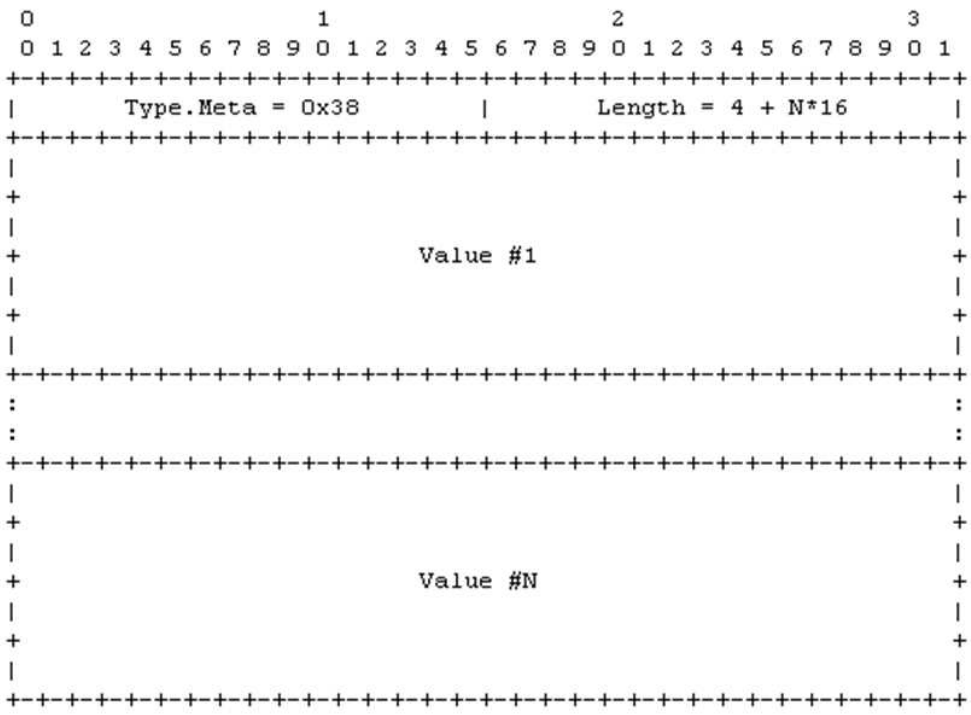

Figure 3.12: Example of simple TLV with 16-Octets Values

As these TLVs are always aligned to 4-octet words, the Length field MUST specify the size of the whole TLV, and padding octets MUST NOT be added.

Opaque TLV Values

An Opaque Value is a sequence of octets that SHOULD NOT be processed by a XBE32 parsing entity, but just be delivered to the upper layer.

The Meta values reserved for Simple XBE32 TLVs carrying Opaque Values are represented in the table showed in figure below.

Meta	TLV Value Description
0x20	Single variable-length opaque Value
0x24	Multiple opaque1 Values
0x28	Multiple opaque2 Values
0x2C	Multiple opaque4 Values
0x30	Multiple opaque8 Values
0x34	Multiple opaque12 Values
0x38	Multiple opaque16 Values

Figure 3.13: Opaque TLV types

String TLV Value

String Values MUST be encoded using UTF-8.

The Meta value reserved for Simple XBE32 TLVs carrying a single String Value is shown in the next figure:

Meta	TLV Value Description
0x21	Single variable-length string Value

Figure 3.14: String TLV types

Boolean TLV Values

Each Boolean Value is encoded with a single octet. A "False" Value is serialized as 0x00, while "True" is encoded as 0xFF. Other values than 0x00 or 0xFF MUST NOT appear as boolean-encoded values.

The Meta value reserved for Simple XBE32 TLVs carrying multiple Boolean Values is shown in the next table:

Meta	TLV Value Description
0x26	Multiple boolean Values

Figure 3.15: Boolean TLV types

Integer TLV Values

Integer Values are signed and MUST be encoded as a two's complement binary number in network byte order (a.k.a. Big Endian, i.e., the most significant byte first).

The Meta values reserved for Simple XBE32 TLVs carrying multiple Integer Values are shown in the table:

Meta	TLV Value Description
0x25	Multiple int8 Values
0x29	Multiple int16 Values
0x2D	Multiple int32 Values
0x31	Multiple int64 Values

Figure 3.16: Integer TLV types

Floating point TLV Values

Floating point Values MUST be encoded as specified in [3].

The Meta values reserved for Simple XBE32 TLVs carrying multiple Floating Point Values are shown in this figure:

Meta	TLV Value Description
0x2E	Multiple float32 Values
0x32	Multiple float64 Values

Figure 3.17: Floating TLV types

3.3 XBE32 Elements

Hierarchical data can be represented as a tree, where each node has an identifier. The "leaf" nodes of the tree are the only ones which are able to carry primitive data values. In XBE32 the nodes of the tree are known as "Elements". Each XBE32 Element has an identifier, that could be a binary one or a human-readable name.

There are two kinds of Elements in XBE32, depending on whether they carry primitive data or not: "Attribute Elements" are the leafs of the tree and carry zero or more Values of a given data type."Complex Elements" on the other hand, cannot carry primitive data, but they are the parents of other XBE32 Elements, that could be Attribute Elements or other Complex Elements themselves.

Furthermore, the so-called "Compact Elements" are encoded inside a single XBE32 TLV, while the optional "Extensible Elements" require two or more TLVs in order to carry their Extensible Names or Identifiers. Moreover, Complex Elements are encoded using Complex XBE32 TLVs, whereas Attribute Elements employ Simple XBE32 TLVs.

Compact Elements

As most network protocols only employ a small set of elements to build their messages, they could be easily encoded with XBE32 by just using Compact Elements, that are encoded with a single TLV and are identified by its binary 16-bit Type field.

Each application/protocol using XBE32 may define its own set of Type values, unless they have been reserved in the base specification of XBE32. Therefore Compact Elements SHOULD employ only the following TLV Meta and Subtype values (with any combination of C and E bits). See figure 3.18.

Meta	Subtype	TLV Description
0x00-0x1F	0x01-0xFE	Compact Complex Element
0x20-0x38	0x01-0xFE	Compact Attribute Element

Figure 3.18: TLV Meta and Subtype table

Compact Attribute Elements MUST employ an appropriate Meta value according to the type of the primitive data carried in their TLV

Values field, as defined in the previous section of this document. For example, an Attribute Element carrying zero or more 32-bit Integer Values may be encoded with one Simple TLV whose Type value is in the 0x2D01-0x2DFE range, with the C and E bits set accordingly to the desired processing rules.

Extensible Elements: Extensible Names and Identifiers

The above mechanism allows a compact representation of binary data and is suitable for the initial definition of the mandatory operations and optional parameters of a simple network protocol. However, a 2-octet Type field may not be enough for truly extensible protocols, as it could be a namespace too small for vendor extensions, experimental operations, or future versions of the protocol.

In order to cope with this limitation, XBE32 implementations MAY also support Extensible Elements. These optional XBE32 Elements are encoded employing multiple TLVs, that are stored inside a XBE32 Complex TLV with a reserved Type value depending on whether the Extensible Element is an Attribute or a Complex one (see figure 3.19).

Meta	Subtype	TLV Description
0x1F	0xFF	Extensible Complex TLV
0x1F	0x00	Extensible Attribute TLV

Figure 3.19: Extensible complex and attributes TLV

Note that C and E bits may have any value, thus, four different Extensible Complex Element TLVs, and other four Extensible Attribute Element TLVs are defined. For instance, an optional Extensible Attribute Element, that should be notified if unknown, must be encoded inside an Extensible TLV with a 0xDF00 Type value.

Each XBE32 Extensible Element MUST have an identifier, that can be a single 4-octet opaque value called Extensible Identifier, or a non-empty UTF-8 string called Extensible Name. The identifier of an Extensible Element MUST be included inside the first inner TLV of the Complex TLV which encodes the Extensible Element. XBE32 has reserved two Simple TLVs to carry Extensible Names and Identifiers. See next figure:

Type	TLV Description
0x21FF	Extensible Name TLV
0x2CFF	Extensible Identifier TLV

Figure 3.20: Extensible TLV names and identifiers

Although the Type field of the upper Extensible TLV does not identify the Extensible Element by itself, its C and E bits are fully meaningful, and MUST specify what measures must be taken if a XBE32 processing entity does not recognize the Extensible Name or Identifier of this Extensible Element, or it just does not support Extensible Elements at all.

Extensible Complex Elements

An Extensible Complex Element is encoded inside an Extensible Complex TLV (Meta=0x1F Subtype=0xFF), that MUST contain a single Extensible Name TLV (Type=0x21FF) or Extensible Identifier TLV (Type=0x2CFF) first, followed by zero or more TLV-encoded XBE32 Elements, that could be Compact or Extensible ones, Attributes or Complex ones, or any combination of them. (See figure 3.21)

The optional "unspecified" length mechanism, when applied to an Extensible Complex TLV, may allow XBE32 processing entities to start encoding and sending partial Extensible Complex Elements before all their sub-elements are known or their data is fully available.

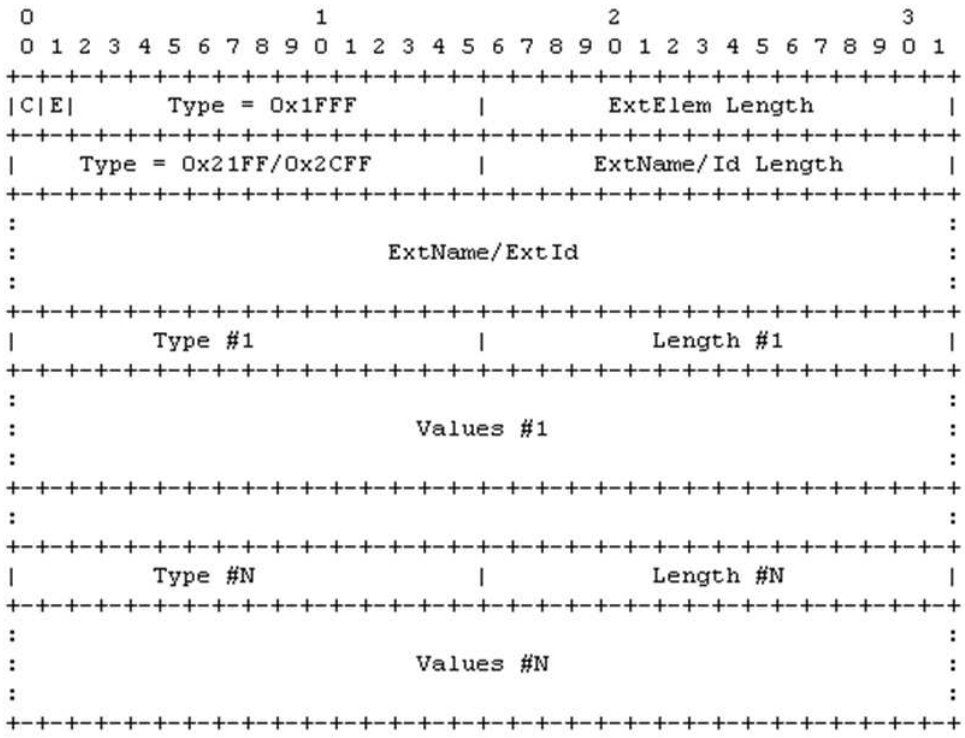

Figure 3.21: Extensible Complex Elements example

Extensible Attribute Elements

The structure of an Extensible Attribute Element is quite similar to an Extensible Complex Element, as it is encoded inside a Extensible Attribute TLV (Meta=0x1F Subtype=0x00), that MUST include at least two TLVs: a single Extensible Name TLV (Type=0x21FF) or Extensible Identifier TLV (Type=0x2CFF) first, followed by one or more TLVs which carry the Values of that Extensible Attribute. (See figure 3.22)

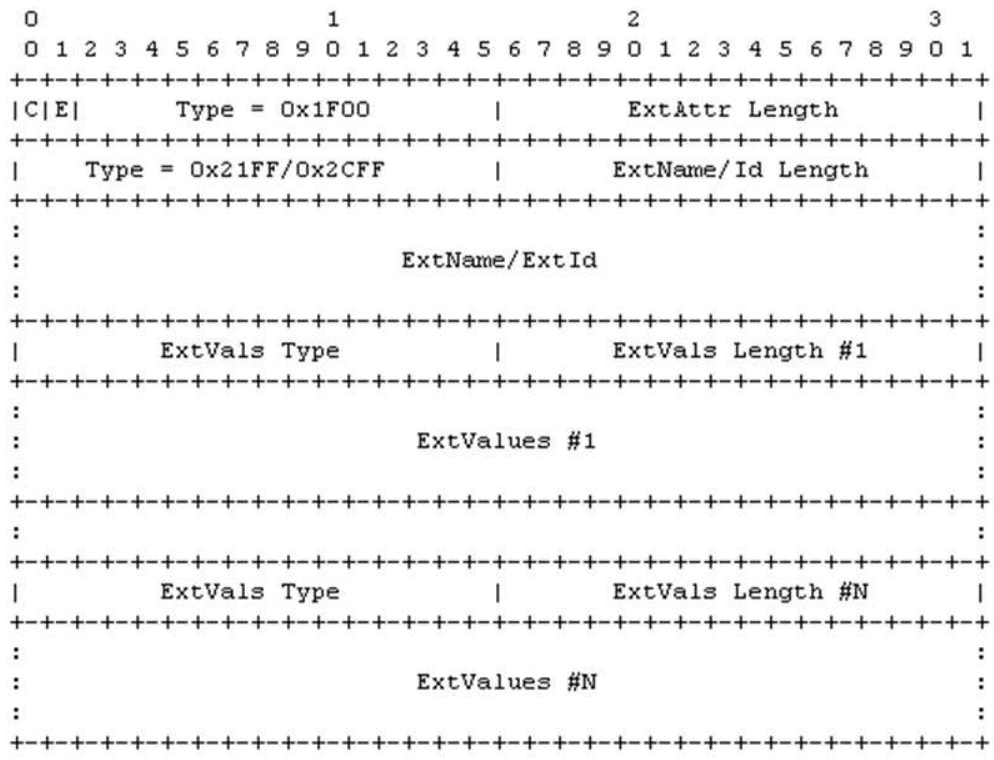

Figure 3.22: Extensible Attribute Elements example

XBE32 has reserved the Type values shown in figure 3.23 for the Extensible Values TLVs.

```
Type    TLV Description
------  ------------------------------------------------
0x2000  Extensible variable-length opaque Value TLV
0x2100  Extensible variable-length string Value TLV
0x2400  Extensible  opaque1 Values TLV
0x2500  Extensible     int8 Values TLV
0x2600  Extensible  boolean Values TLV
0x2800  Extensible  opaque2 Values TLV
0x2900  Extensible    int16 Values TLV
0x2C00  Extensible  opaque4 Values TLV
0x2D00  Extensible    int32 Values TLV
0x2E00  Extensible  float32 Values TLV
0x3000  Extensible  opaque8 Values TLV
0x3100  Extensible    int64 Values TLV
0x3200  Extensible  float64 Values TLV
0x3400  Extensible opaque12 Values TLV
0x3800  Extensible opaque16 Values TLV
```

Figure 3.23: Dissected TLV type

If several Extensible Values TLVs are present, all of them MUST have the same Type value, depending on the data type of the Extensible Attribute Element. It is RECOMMENDED to encode all the Values of an Extensible Attribute in a single Extensible Values TLV, whenever it is possible. Nevertheless, a XBE32 processing entity SHOULD concatenate, keeping the received order, all the Values fields of all the Extensible Values TLVs forming an Extensible Attribute Element. For instance, multiple Extensible string Value TLVs should be appended to form a single variable-length String Value, whereas several Extensible int32 Values TLVs would generate a single array of Integer Values.

The encoding of Extensible Attributes with multiple Extensible Values TLVs, paired with the optional "unspecified" length mechanism, may allow XBE32 Extensible Attributes to carry a single Value or a list of Values longer than the 65532 octets limit of Compact Attributes.

Chapter 4

XBE32 DESIGN AND IMPLEMENTATION

This chapter explains how the implementation and design of XBE32 has been done. It shows the mechanisms and structures used to develop it, and why some design decisions have been taken.

As it can be seen in the previous chapter, XBE32 handles two different concepts: Elements produced by the application and TLVs that encode such informations. Therefore it seems immediate, that the implementation of the encoding system follows a two tier software architecture, in which the first layer is the one corresponding to the processing of the TLVs, and the second one is in charge of managing elements in the XBE32 encoding language. This two layer application has its lower layer in the TLV processing layer, while, the upper layer, the element processor, takes advantage of the first.

In addition to these different global functionalities, we have decided that is important to implement a way to define all the different element types employed by the user, so another functionality has been added in order to manage a dictionary. Obviously the use of this component is optional, since the goal of this whole project was to build a library for encoding XBE32, and the last component is just an item to make the user life easier.

Despite the two layers have their functionalities very delimited, the main operations in both of them remain the same: write and read (whatever TLVs or elements). In the first case, it is necessary to put and find the limits of any TLV, check the types of each one, and to know where we must stop processing

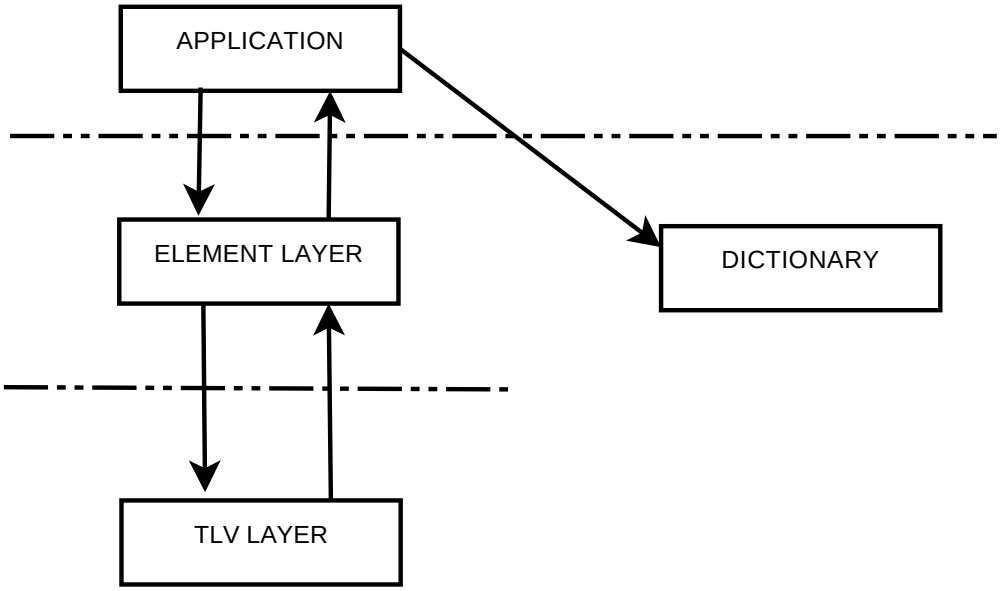

Figure 4.1: Library structure

them. For the elements the proceedings are done in different way, but the goals are the same: write whole items and retrieve them without error. Therefore each one of the layers of the application can be split in two: The writer and the reader.

As any C Library this XBE32 library is just a set of functions that implement some part of the encoding/decoding process. Since both processes require several steps, it is necessary to mantain some state. Therefore, instead of relaying on global variables that would allow just a single encoding/decoding flow per process, the state has been stored inside opaque structures that are passed to all XBE32 library functions. From the user's point of view, these structures model the XBE32 encoding/decoding process.

4.1 TLV layer

This layer handles the serialization of data on TLVs, and the parsing of simple and complex TLVs, including the ones with unspecified length that are closed with an End-of-TLV.

The writer: Building a TLV

To build the TLV a structure in which all the necessary items to write the TLV are present has been thought up. It is oriented to handle the writing in different buffers if needed, an have an account of the open TLVs. In addition to that, it handles the delivery of error messages. This structure is the writer itself, and it is defined as follows:

```
struct xbe32_tlv_writer {
  unsigned char* buffer_start;
  unsigned char* buffer_end;
  unsigned char* buffer_ptr;
  writer_stack_t * open_tlvs;
  unsigned long bytes_counter;
  int num_end_of_tlv;
  char * error_msg;
  int error_code;
};
```

Three different members to handle the buffer;
buffer_start: points the start of the current buffer. buffer_end: points the end of the current buffer. buffer_ptr: points to the next byte to be written in the buffer.

Two members to track the open TLVs since the total length of a complex TLV is unknown until all its inner TLVs have been written. Only then the length of the complex TLV can be filled. If the buffer is flushed before the TLV length can be determined, the TLV has a zero length which means that the appropriate number of End-of-TLV must be inserted;
open_tlvs: a pointer to a stack which saves the open TLVs. num_end_of_tlv: saves the number of open TLVs in case a change of the buffer takes place.

Three members of general purpose;
bytes_counter: counts the number of bytes that have been written during the process until the current moment. error_code: carries the last error code. In case there is none, is set to 0. error_msg: in case the error_code member has a valid code, this member carries the written notification to that error.

The application has been structured in layers in order to achieve transparency, so the writer. To take fully advantage of the writer, it is necessary to

access it through primitives. These primitives are explained below.

- xbe32_tlv_writer_t * xbe32_tlv_createWriter (unsigned char * buf, int len): creates and initiates the writer. It allocates memory for the writer structure which will perform the task of writing the encoded message through its buffers. It associates a buffer to the writer; sets the different pointers to its correspondent locations, buffer_start and buffer_ptr point to the start of the newly assign buffer; makes buffer_end points to the last byte of the buffer (by adding to the buffer_start the _length of the buffer) allowing the upper application to know how far it can write. Sets the null values to the other variables. It returns the initialized buffer.

- void xbe32_tlv_setWriterBuffer (xbe32_tlv_writer_t * writer, unsigned char * buf, int len): changes the buffer in case the previous one is discarded (because it is full or any other reason). To do that, it is necessary to set the pointers in the corresponding places. buffer_start and buffer_ptr must point to the begin of the buffer; buffer_end again is calculated as the adding of the length of the new buffer to buffer_start. The rest of the members of the writer not need to be initialized.

- int xbe32_tlv_flush (xbe32_tlv_writer_t * writer): saves the number of open TLV in the member num_end_of_tlv. Since in this function not a single byte is written the returned value is 0.

- void xbe32_tlv_destroyWriter (xbe32_tlv_writer_t * writer): frees the memory allocated for the writer.

- int xbe32_tlv_openTLV (xbe32_tlv_writer_t * writer, uint16_t type): writes the header of a complex TLV. The writer and the type of the TLV to write are passed as arguments. Once the function has checked if there is space available to do the writing, the type is written in Internet byte order, as the length is unknown for the moment, it is set to 0 (coded in Internet order as well). Writer member buffer_ptr advances the size of the bytes written. This number of bytes are returned by the function.

- int xbe32_tlv_writeTLV (xbe32_tlv_writer_t * writer, uint16_t type, void * vals, int vals_size): writes the payload of the TLV. First checks if there is enough space available, if not propagates an error. If there is space available, checks the type of the TLV and writes it in the proper way depending on the type. Once this has been made, buffer_ptr advances the

total length of the payload (including the padding), and the total number of bytes written during the function is returned.

- int xbe32_tlv_closeTLV (xbe32_tlv_writer_t * writer): closes the last open TLV (This is just applied to complex ones). To do that, takes the first TLV saved on the stack, and, if the stack is not empty, or if the num_end_of_tlv member is not 0, it writes the length of the TLV in the corresponding field or an end-of-data TLV, indicating that is the end of a TLV of unspecified length.

The first three functions, are in charge of handling the writer structure. To initiate the writer structure members, allocate and free the memory. The last four handle the writing of the TLVs. The first of them, xbe32_tlv_openTLV, creates the header for complex TLV. The complement to that function is xbe32_tlv_closeTLV that is in charge of closing open complex TLVs. To write simple TLVs, xbe32_tlv_writeTLV is used, and xbe32_tlv_flush is used to manage the open TLVs in case there is a change of buffer (in case the buffer is changed, the pointers are of no use).

The reader: Processing a TLV

This layer is not supposed to return nothing valuable for the final user since it returns not whole elements that the user can understand but TLVs. In spite of this, perhaps can be of some interest to explain how this TLV must to be interpreted. This layer just return TLV to the next one. The TLV returned can be whole TLVs, so to speak, a TLV with a header and some contents, or it can return the header of a complex TLV. In the last case, this TLV signals the start of a complex TLV which in the layer above can be translated as a complex element or even an extensible element.

As the application must be able to write a TLV, it must be able to decipher it too once this is written. Just to do this, the reader structure (parser) has been created, it is the mean to read the TLVs once these have been coded. The reader structure is defined as follows:

```
struct xbe32_tlv_parser{
  unsigned char* buffer_start;
  unsigned char* buffer_end;
  unsigned char * parser_ptr;
```

```
  unsigned long bytes_counter;
  parser_stack_t * open_tlvs;
  char * error_msg;
  int error_code;
};
```

As in the structure of the writer, three different members handle the buffer;
buffer_start: points to the start of the current buffer.
buffer_end: points to the end of the current buffer.
parser_ptr: points to the next byte to be read in the buffer.

Three members of general purpose;
bytes_counter: keeps track of the bytes read from the buffer until the moment. error_code: carries the last reading error code. In case there is none, is set to 0. error_msg: in case the error_code member has a valid code, this member carries the written notification to that error.

One member to process the open TLVs;
open_tlvs: a pointer to a stack which saves the open TLVs. With this pointer the application knows when the complex TLV is closed and thus when to finish the reading. It is also required to remember the TLV that a n End-of-TLV is closing.

Again, as with the writer, the parser is accessed only through primitives to achieve the goal of transparency. The functions created to do this are the following:

- xbe32_tlv_parser_t * xbe32_tlv_createParser (unsigned char * buf, int len): Initializes and creates the parser structure. With this structure the library will be able to decode the XML messages.
  It associates a buffer to the parser; sets the different pointers to its correspondent locations, buffer_start and parser_ptr point to the start of the newly assign buffer, makes buffer_end points to the last byte of the buffer (by adding to the buffer_start the length of the buffer) allowing the upper application to know the point to finish the reading. Sets the null values to the other variables. It returns the initialized buffer.

- void xbe32_tlv_setParserBuffer (xbe32_tlv_parser_t * parser, unsigned char * buf, int len): With this function, the application changes the buffer

which is currently being read. In the case there is a change of buffer, this function allows the application to change the buffer to read in case the last one is ended and it is necessary to continue reading another one.

- void xbe32_tlv_destroyParser (xbe32_tlv_parser_t * parser): This function erases the parser structure. To do it, it frees the memory allocated for the parser.

- uint16_t xbe32_tlv_getType (xbe32_tlv_t * tlv): It takes the currently processed TLV and returns its type. This type is useful to process the TLV (it allows to know if it is a complex TLV or a simple one, end if it is simple, the type of the data inside the TLV).

- uint16_t xbe32_tlv_getLength (xbe32_tlv_t * tlv): Returns the length of the current TLV. This length is the one correspondent to the length of the TLV including the header, payload of the TLV, but not the padding.

- bool xbe32_tlv_getContinueFlag (uint16_t type): Indicates to the application if the processing must continue or not in case an error occurs. It returns a boolean flag indicating if the upper application must continue the processing in case there is a failure.

- bool xbe32_tlv_getErrorFlag (uint16_t type): Indicates to the application if there must be a notification of an error on the message. It returns a boolean flag indicating to the upper application whether an error message should be sent back to the source.

- uint16_t xbe32_tlv_getMeta (uint16_t type): It takes the proper type of the TLV. This is, the type without the error and the continue flag.

- bool xbe32_tlv_isComplex (uint16_t type): Gives the user information about the nature of the TLV. Returns a boolean value which indicates that current TLV is a complex one.

- int xbe32_tlv_getNumValues (xbe32_tlv_t * tlv): This function returns an integer that returns the number of basic values (e.g Floats, Integers, Strings...) presents in the TLV.

- int xbe32_tlv_getValuesLength(int length): This function returns an integer indicating the length of the payload of the TLV.

- unsigned char * xbe32_tlv_getValues (xbe32_tlv_t * tlv): This function returns a pointer to the values stored inside the TLV.

- bool xbe32_tlv_endOfBuffer(xbe32_tlv_parser_t * parser): Checks if the buffer which is being currently read has reached the end. Returns a boolean value.
- int xbe32_tlv_paddedLength(int l): This function returns the length of the padding attached to the payload.
- xbe32_tlv_t * xbe32_tlv_nextTLV (xbe32_tlv_parser_t * parser, bool * closed): The most important function in the reader side. It orchestrates the reading of the bytes on the buffer and decides how many of them form a TLV. Once this has been done, the bytes correspondent to the TLV are stored in a TLV structure and passed to the upper layer.

The first three functions, are in charge of handling the parser itself. To initiate the members and allocate and free the memory. The next handle the reading of the TLVs. The first functions are in charge of handle aspects of the TLVs as the length, type, payload, etc. All these fuctions are applied to the last TLV read by means of the nextTLV function. The last one, xbe32_tlv_nextTLV is in charge to direct the other functions in order to structure the TLV in certain variables to pass the information to the upper level.

One of the most relevant functions in this library, in spite it is quite simple in its implementation, is xbe32_tlv_openTLV, which opens a complex TLV. Next we are going to show a flowchart 4.2 that describes the function:

- First, checks if there is enough space in the buffer to write the TLV's header (just type and length).
- If there is enough space it writes the type and sets the length to 0 (the length up to this point is unknown). After that, it puts the writer pointer four bytes forward (just the number of bytes that have been written).
- If there is not enough space, it sets an error message in the error_message member and its corresponding error code in the error_code member. Both will be propagated depending on the values in 'e' and 'c' flags respectively.
- Finally, the application will return the number of written bytes.

Once a complex TLV has been open, the protocol requieres it to be closed. To do this, the application implements xbe32_tlv_closeTLV. The chart corresponding to this function is showed in the figure 4.3:

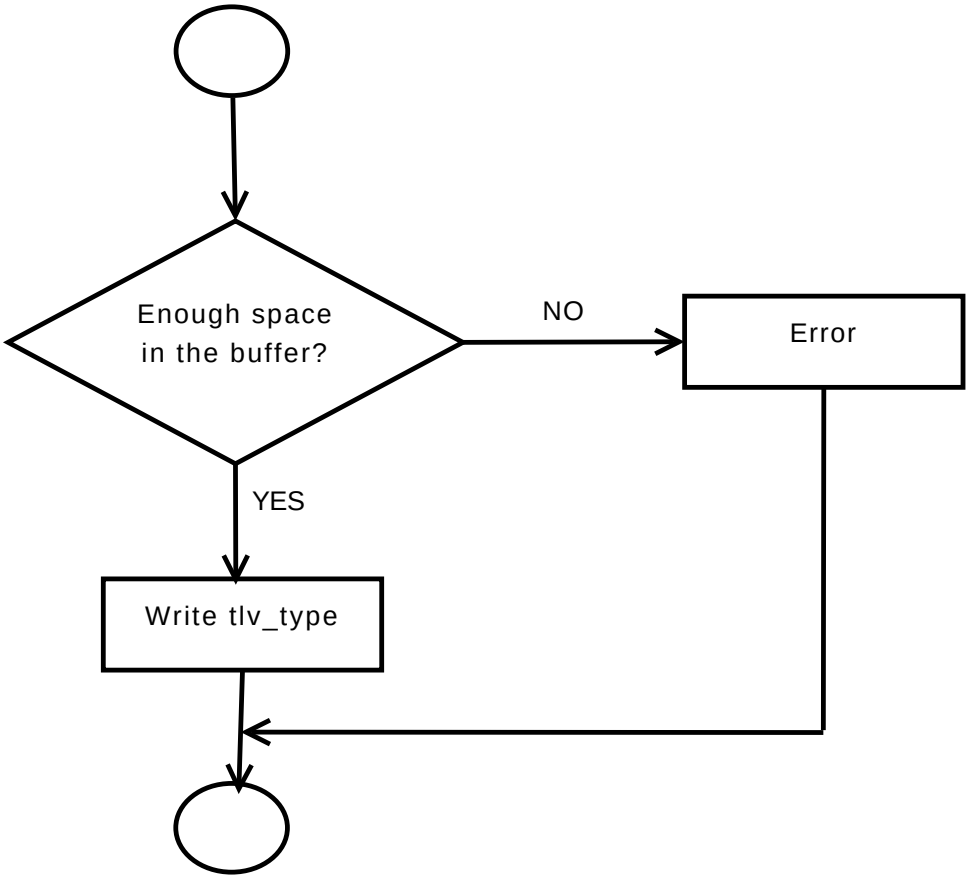

Figure 4.2: Opening of a complex TLV

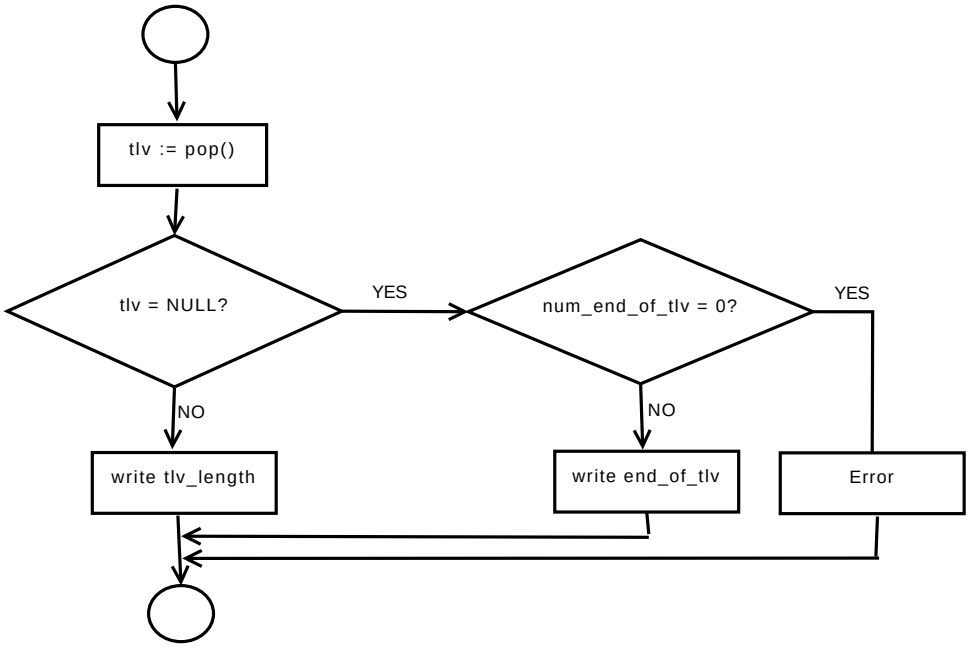

Figure 4.3: Closing of a complex TLV

- The function removes the last TLV from the stack.

- Checks if the obtained TLV is NULL. If it is checks if the member of the writer structure num_end_of_TLV is equal to 0. If num_end_of_TLV is equal to 0, it sets an error message in the error_message member and its corresponding error code in the error_code member. As in the previous flowchart, both will be propagated depending on the values in 'e' and 'c' flags respectively.

  - if num_end_of_TLV is not equal to 0, then the function writes an End_of_TLV header indicating that the end of the last open complex TLV must be assumed.

- If the obtained TLV is not NULL, the function writes the corresponding length in the complex TLV's header.

As it has been said, xbe32_tlv_nextTLV is the most important function in this layer. Figure 4.4 shows the flowchart for the function.

Since it is the most important for the TLV layer, we are going to proceed to explain the steps that it follows through the Figure 4.4:

- First, it checks if the application has reached the end of the buffer. In this case, it is not possible to continue the parsing and the user has to take care of changing the buffer, meanwhile, the function exits returning a null value.

- In case the buffer end has not been reached, next, the function checks if the current point of the buffer matches up with the end of a complex TLV stored in the parser stack. In this case, this means that a complex TLV has reached its end in the previous xbe32_tlv_nextTLV iteration, and thus, it must be closed. If this is the case, the TLV variable takes the value of the last item stored in the stack, through a pop function, and it is returned. The function must use the isClosed function to know whether the returned TLV is an old TLV being closed or a new one.

- The function will take the next TLV data to be processed (this data which actually is written in the buffer, will be stored in a TLV variable through a casting operation). If the previous case is negative, then, the function checks if there is enough space in the parser buffer to process the TLV.

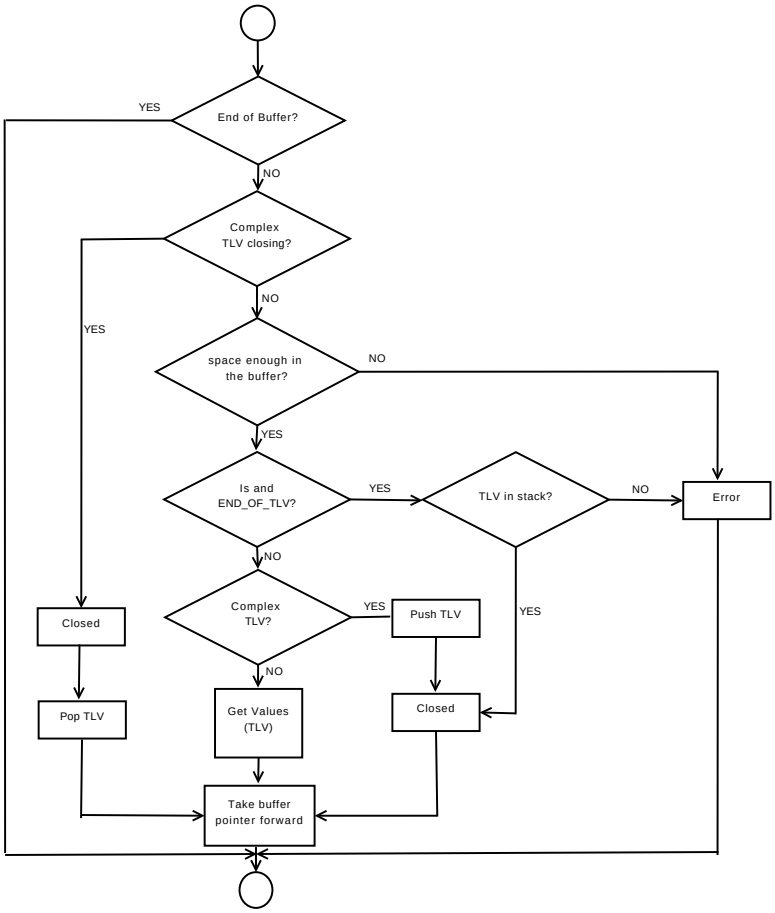

Figure 4.4: Next TLV process

- Next, the function will check if the taken TLV (casting) is a complex one (since it has been checked that the current TLV is not being closed, this complex TLV must be about to be open). If it is, it will be stored in the stack (to know that it must be closed in the future), and the flag saying that it is not to be closed will be set to true.
- In case the TLV is not a complex one, the function will sort the values in the TLV (in case it is necessary because the computer byte order) and will take the parser pointer just forward enough for the processing of the next data.
- The function will return the TLV.

4.2 XBE32 Element layer

The writer: Building an XBE32 element

Once all the processes related to TLV has been explained, it is time to go to a layer above and talk about the elements. As with the TLV, the implementation of the elements has been split in two different parts: one corresponding to the writing of the elements, and other corresponding to the reading.

The writing process for xbe32 elements is quite simple once the TLV primitives for the writing have been defined. As in the previous layer, the elements are processed through a writer structure. This structure is the same as with the TLV, but for this layer its type is renamed as xbe32_writer.

The primitives handling the writer structure are the following:

- xbe32_writer_t * xbe32_createWriter (unsigned char * buf, int len): This function creates a TLV writer (xbe32_tlv_writer_t) to the element writer type (xbe32_writer_t), and returns an initialized writer variable calling to xbe32_tlv_createWriter.
- void xbe32_destroyWriter (xbe32_writer_t * writer): Deallocates the memory belonging to the writer variable. It does it by calling to xbe32_tlv_destroyWriter function.
- void xbe32_setWriterBuffer (xbe32_writer_t * writer, unsigned char * buf, int len): Changes the buffer which is being currently written for

another one once the first has reached its end. To do that, the function calls to xbe32_tlv_setWriterBuffer.

- void xbe32_flush (xbe32_writer_t * writer): Calls to xbe32_tlv_flush in order dump the contents to the upper application environment (empties the buffer dumping its contents to be immediately processed).

- int xbe32_openElement (xbe32_writer_t * writer, uint16_t type): Opens a compact XBE32 complex element with the specified type (using xbe32_openTLV).

- int xbe32_writeAttr (xbe32_writer_t * writer, uint16_t type, void * vals, uint32_t length): Writes a simple XBE32 attribute. To do that just writes a single TLV (through xbe32_tlv_writeTLV function). In case the size of the TLV to write exceeds 65532 bytes, it returns an error message.

- int xbe32_closeElement (xbe32_writer_t * writer): Closes a compact XBE32 complex element.

- int xbe32_openExtElement (xbe32_writer_t * writer, uint32_t id, char * name): Opens an extensible element. To do this, first it is necessary to open a complex TLV which will contain the inner elements (through xbe32_tlv_openTLV), and after that, to write an identifier (a TLV containing the ID for the item) for the extensible element, this identifier can be a name or a numeric identifier (to write the identifier xbe32_tlv_writeTLV will be called).

- int xbe32_writeExtAttr (xbe32_writer_t * writer, uint32_t id, char * name, int vals_type, void * vals, int length): writes an extensible attribute. To do that, first opens the extensible attribute (calling to xbe32_tlv_openTLV), second identifies the extensible element (it inserts a TLV with the name or identifier of the extensible attribute calling to xbe32_tlv_writeTLV), third writes the attribute contents (through xbe32_tlv_writeTLV function) and finally, closes the extensible attribute (by closing the TLV that contains both the name and contents of the extensible attribute through xbe32_tlv_closeTLV).

- int xbe32_closeExtElement (xbe32_writer_t * writer): Close an complex element that has been previously opened (by calling to xbe32_tlv_closeTLV). It must be said that the implementation of xbe32_closeElement and xbe32_closeExtElement is the same, but the latter has been included in the implementation for symmetry reasons.

The process of writing elements with this library is simple [3]. It only consists on writing the primitives to write the different elements in a sequential way which describes the order inside the elements that the user wishes to write.

The reader: Processing a XBE32 element

The reading of XBE32 elements is a little bit more complicated than the writing. First of all, the structure needed to do all the process is basically the same as in the layer below, but replacing the TLV stack with a XBE32 Element stack. In addition to that, it is necessary to declare a new structure where the items will be stored. Both structures are going to be explained in the next paragraph.

As it has been said before, a new structure has been created in this layer to return all the information that form the read elements. This structure is defined as follows:

```
struct xbe32_elem {
  char * name;
  uint32_t id;
  uint16_t flags;
  int valuesType;
  int valuesNum;
  void * values;
};
```

The meaning of the fields are explained bellow;
name: represents the name of an extensible element (or attribute). If there is no name, it is set to NULL.
id: represents the id of an element (no matter it is a compact/extensible simple/complex one).
flags: Contains the whole type of the element, in order to get later the meta flags (continue and error).
valuesType: Contains the basic type of the element, without the meta component (flags continue and error).
valuesNum: Contains the number of values in the element.
values: A pointer to the content of the element.

[3] See appendix B

The basis of the parser structure is the one present in the layer below but with an additional field. This field will be the stack, necessary to save all the possible elements inside complex elements. The structure of the stack is the following:

```
struct xbe32_stack{
  xbe32_elem_t * item;
  xbe32_stack_t * next;
};
```

The meaning of these fields are;
item: represents the element which needs to be saved in order to be part of a bigger and more complex element.
next: link to the next element present in the stack.

The whole reading processing is carried out mainly by the xbe32_nextElement() function. As in the lower layer, it is the function in charge of orchestrating the other primitives to compose the element correctly. Roughly, what this function does is to take TLVs, and in case the TLV is a complex one (what means that the element currently processed is complex, or extensible) ask for more TLV to get the attributes or elements that compose the processed element, if is not a complex TLV, that means that it is a simple element or a part of a complex one, so returns the values contained in that TLV. The rest of the functions declared in the parser section are functions that performs tasks for xbe32_nextElement() or functions that handle the parser structure and its buffer operations.

What the user will obtain of this layer are different elements. This elements will contain data (a whole element with its id/name and values), or just a id for complex elements. In the latter case, the id/name are sent to indicate the user that an item has been open and other elements are nested inside or that an element is closing. Both cases can be differentiated because one of the parameters in the function is a reference and, in case this parameter is set to true it means that the current element is closing. As can be seen, this manoeuvre is quite similar to the one performed by xbe32_tlv_nextTLV() in the previous layer.

The primitives to handle the parser and the whole process of reading XBE32 elements are the next:

- xbe32_parser_t * xbe32_createParser (unsigned char * buf, int len): Allocates memory for the parser structure and initializes it.
- void xbe32_setParserBuffer (xbe32_parser_t * parser, unsigned char * buf, int len): Performs a change of buffer under user request.
- void xbe32_destroyParser (xbe32_parser_t * parser): Deallocates the memory of the parser under user request.
- xbe32_elem_t * xbe32_nextElement (xbe32_parser_t * parser, bool * closedElem): Returns the next XBE32 element on the buffer.
- bool xbe32_isExtensible (uint16_t type): Returns a boolean that indicates if the specified element is extensible or not. Takes the type as parameter.
- int xbe32_getFlags (xbe32_elem_t * elem): Returns the meta flags of the XBE32 element type. Takes as argument the element.
- int xbe32_getNumValues (xbe32_elem_t * elem): Returns the number of values inside an XBE32 element. Takes as argument the element.
- int xbe32_getId (xbe32_elem_t * elem): Returns the Id of the an XBE32 element whether this is extensible or not. Takes as argument the element.
- char * xbe32_getName (xbe32_elem_t * elem): Returns the name of an XBE32 element in case it has one. Takes as argument the element.
- bool xbe32_isComplex (xbe32_elem_t * elem): Returns a boolean that tells if the element is complex or not. Takes as argument the element.
- int xbe32_getValuesType (xbe32_elem_t * elem): Returns the basic type of the element. Takes as argument the element.
- xbe32_getValues: Returns a void pointer to the contents of the XBE32 element. Takes as argument the element.
- void * xbe32_getNumValues (xbe32_elem_t * elem): Returns the number of values of a certain type inside an XBE32 element. Takes as argument the element.

As with the lower layer, one function is in charge of orchestrate the rest (not in the writer since all the functions must be directly selected by the user). With this flowchart, xbe32_nextElement will be explained:

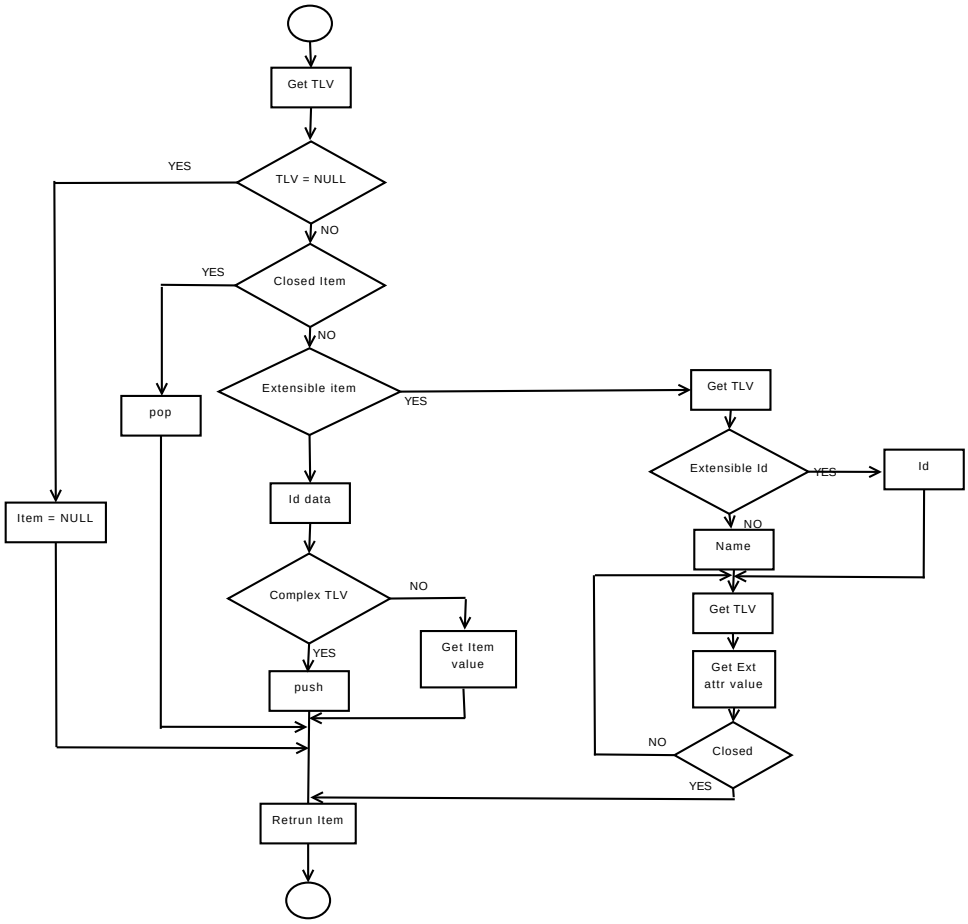

Figure 4.5: Next Element process

- First of all, the function takes the TLV provided by the lower layer (xbe32_nextTLV).
- After that, the function checks if the TLV with the possible contents for the element has a null value. In that case, the item to be returned will be set to NULL, and with its return the function finishes its execution.
- If the TLV provided by the lower layer is not null, the function will check next if the TLV is signaling the closing of the element. In case the latter is true, a pop of the previous stored data for the element will be done, and that data will be stored in an element variable that next will be returned finishing the execution of the function.
- If the TLV does not tell that the end of the element has come, the next checking the function will perform is about extensibility. If the TLV contains a type saying that the element is extensible, the following steps are to follow:
  - In case the TLV shows that the element is extensible, immediately, the function takes another TLV from the lower layer.
  - Checks if the content of the TLV is an identifier or a name, and in both cases though through different proceedings, obtains them.
  - After that, another TLV is taken from the lower level. This TLV is supposed to have the values of the element, which are taken.
  - The previous steps are repeated many times as needed until all the values in the extensible element are obtanied.
  - Once the extensible element is complete, the element value is returned and the function exited.
- In case the element is not extensible, some of the data which the TLV (the one addressed in the third point) carries are related to identification characteristics such as; type, id/name and flags. The next action by the function will be to store this characteristics in the element variable.
- Next, the function will find out if the TLV is complex or not. If it is not the case, the data from the TLV will be extracted and placed into the element variable, and with this data, the element will be returned, finishing the function.
- If the TLV is a complex one (its contents are empty, but the field tells that a whole TLV which represent physically the element is composed by more TLV), the element variable will be initialized, and through a push

function the element will be stored awaiting for the rest of the components in the stack. In addition to that, the element will be delivered to the client application in order the latter knows that an extensible element has been opened. With this last action, the function exists.

The dictionary: Giving sense to the elements

Finally to make the implementation of the protocol, a third complement must be added by the upper function (the one susceptible to use XBE32 in its communication), that is the implementation of the dictionary. The dictionary is needed to translate some of the complex TLV to elements with a meaning to the so called upper application.

This component is composed by several items that represent each complex identifier in the upper application namespace. To implement the dictionary utility, a main structure has been created to store the different words, this structure will be the base of a dynamic list (a node) and it is defined as follows:

```
struct xbe32_dictionary {
  uint16_t type;
  char * name;
  xbe32_dictionary_t * next;
};
```

The meaning of the different files of the structure is the next;
type: it will contain the type of the represented element.
name: it will contain the name of the represented element.
next: it will contain a pointer to the next XBE32 represented in the dictionary.

- xbe32_create_item: Creates the node about to be inserted in the dictionary. As parameters takes the name of the item and the type.
- xbe32_dictionary_loadDictionary: It is a function that creates a list (which will become the dictionary) from a file. On that list it will be all the words belonging to the namespace of the application that is going to use the XBE32 library.

- xbe32_dictionary_lookup: With this function it is possible to look up any ID in the dictionary and return the correspondent word.
- xbe32_dictionary_cover: This function is implemented to facilitate a complete list of words in the dictionary. It covers the list and shows the items one by one.
- xbe32_dictionary_free: With this function, the memory allocated to create the dictionary is deallocated.

Despite a dictionary is not strictly part of the XBE32 library, it has been decided to implement it in order to facilitate the user the use of the library. It is helpful because the most part of the items has a recognizable name (more human readable) and through the dictionary utility it is possible to retrieve it.

4.3 Usage examples

In this section a couple of examples of how the library works are going to be included. In the first one, the encoding in XBE32 of an XML code example procedent from the document "Overview of the eXtensible Service Discovery Framework" [2]:

```
<service>
  <id>8e9d7823-d5ac-497c-91d0-fb07ea0c3fb2</id>
  <serviceState>
    <metaInfo>
      <stateTimestamp>f85444f4eb</stateTimestamp>
    </metaInfo>
    <selectState>
      <workload>0</workload>
    </selectState>
  </serviceState>
  <serviceMainInfo>
    <serviceType>
      <type>printer</type>
    </serviceType>
    <alias>Alice's printer</alias>
    <selectInfo>
```

```
            <policies>Least Used (0x0002)</policies>
            <weight>14</weight>
          </selectInfo>
          <printer:color>false</printer:color>
          <printer:duplex>true</printer:duplex>
        </serviceMainInfo>
        <serviceLocation>
          <inet>
            <ipv4Addrs>169.254.85.139</ipv4Addrs>
            <ipv6Addrs>fe80::202:b3ff:fe3c:da7a</ipv6Addrs>
          </inet>
          <protocol>
            <name>ipp</name>
            <transPorts>tcp/631, sctp/631</transPorts>
          </protocol>
          <protocol>
            <name>lpr</name>
            <transPorts>tcp/515, sctp/515</transPorts>
          </protocol>
        </serviceLocation>
        <serviceAddInfo>
          <model>Acme Laser Printer 2000</model>
          <modelURL>http://www.acme.com/printers/lp200.html</modelURL>
        </serviceAddInfo>
      </service>
```

Encoding example

Following, the C code corresponding to the application written to encode the XML code presented in the previous section:

```
int main()
  xbe32_dictionary_t * dictionary;
  xbe32_dictionary_loadDictionary( "dictionary.txt", &dictionary );
  //Declaring and loading the dictionary utility

  xbe32_writer_t * writer = xbe32_createWriter( buffer, 3500 );
  //Creating the writer
```

```
  xbe32_openElement( writer, service);
  //Opens the main compact XBE32 complex element

  xbe32_writeAttr( writer, 0x3811, (void *) id, 16 );
  //writes a simple XBE32 attribute

  xbe32_openElement( writer, servicestate);
  xbe32_openElement( writer, metainfo);
  //Opens two compact XBE32 complex elements (servicestate and metainfo)

  xbe32_writeAttr( writer, 0x3101, (void *) &statetimestamp, 8 );
  //writes a simple XBE32 attribute (statetimestamp)

  xbe32_flush( writer );
  //dumps the contents of the buffer in which it is being written to
the upper application (this)

  xbe32_closeElement( writer ); //closes metainfo
  //closes the last element which has been open

  xbe32_openElement( writer, selectstate);
  xbe32_writeAttr( writer, 0x2E12, (void *) workload, 4);
  xbe32_closeElement( writer );//selectstate
  //opens one compact XBE32 complex element, selectstate. Writes a simple
XBE32 attribute, workload.
  //And closes the last open compact XBE32 complex element, selectstate.

  xbe32_closeElement( writer );//servicestate
  //Closes the first compact XBE32 complex element pending in the stack
(the last one opened), servicestate.

  xbe32_openExtElement( writer, servicemaininfo, NULL);
  xbe32_openExtElement( writer, servicetype, NULL);
  xbe32_writeAttr( writer, 0x2112, (void *) type, 7 );
  //Opens two extensible XBE32 complex elements: servicemaininfo and servicetype.
Writes a simple XBE32 attribute, type.
```

```
  xbe32_closeElement( writer ); //servicetype
  xbe32_writeAttr( writer, 0x2148, (void *) alias, 15 );
  //Closes the last XBE32 complex element, servicetype (which is also extensible),
and write another simple XBE32 attribute, alias.

  xbe32_openExtElement( writer, selectinfo, NULL);
  xbe32_writeAttr( writer, 0x2845, (void *) policies, 2 );
  xbe32_writeAttr( writer, 0x2D45, (void *) weight, 4 );
  //Opens another extensible XBE32 complex element, select info and write
two simple XBE32 attributes: polocies and weight.

  xbe32_closeElement( writer ); //selectinfo
  xbe32_writeExtAttr( writer, 0, "color", STRING_TYPE, (void *) "false", 5 );
  xbe32_writeExtAttr( writer, 0, "duplex", STRING_TYPE, (void *) "true", 4 );
  xbe32_closeElement( writer ); //servicemaininfo
  //Closes selectinfo, and write two simple XBE32 attributes: color and
duplex. Closes servicemaininfo

  xbe32_openExtElement( writer, servicelocation, NULL);
  xbe32_openExtElement( writer, inet, NULL);
  xbe32_writeAttr( writer, 0x2C15, (void *) ipv4addrs, 4 );
  xbe32_writeAttr( writer, 0x3816, (void *) ipv6addrs, 24 );
  xbe32_closeElement( writer ); //inet
  //Opens two extensible XBE32 complex elements: servicelocation and inet.
Writes two simple XBE32 attributes: ipv4addrs and ipv6addrs.
  //Closes inet

  xbe32_openExtElement( writer, protocol, NULL);
  xbe32_writeAttr( writer, 0x2178, (void *) name1, 3 );
  xbe32_writeAttr( writer, 0x2C77, (void *) transports1, 17 );
  xbe32_closeElement( writer ); //protocol
  //Opens one extensible XBE32 complex element, protocol. Writes two
simple XBE32 attributes: name1 and transports1.
  //Closes protocol

  xbe32_openExtElement( writer, protocol, NULL);
  xbe32_writeAttr( writer, 0x2178, (void *) name2, 3 );
  xbe32_writeAttr( writer, 0x2C77, (void *) transports2, 17 );
```

```
  xbe32_closeElement( writer ); //protocol
  xbe32_closeElement( writer ); //servicelocation
  //Opens one extensible XBE32 complex element, protocol. Writes two
simple XBE32 attributes: name2 and transports2.
  //Closes protocol and servicelocation which was open two blocks ago.

  xbe32_openExtElement( writer, serviceaddinfo, NULL);
  xbe32_writeAttr( writer, 0x2189, (void *) model, 23 );
  xbe32_writeAttr( writer, 0x2177, (void *) modelurl, 39 );
  xbe32_closeElement( writer ); //serviceaddinfo
  //Opens one extensible XBE32 complex element, serviceaddinfo. Writes
two simple XBE32 attributes: model and modelurl.
  //Closes serviceaddinfo.

  xbe32_closeElement( writer ); //service
  //Closes service, the first compact XBE32 extensible element opened
for the application.

  //This section below is just to have the output: the encoded text
  buffer_len = 436;
  printf("buffer[%d]:", buffer_len);
  for (i=0; i<buffer_len; i++) {
    if (i%4 == 0) {
      printf("\n");
    }
    printf("%.2x ", buffer[i]);
  }
}
  bool closed = false;
  xbe32_elem_t * item = NULL;
  int item_length = 0;
  unsigned char * vals;

  xbe32_parser_t * parser = xbe32_createParser( buffer, buffer_len );
  //The parser structure is created

  do {
    item = xbe32_nextElement( parser, &closed );
```

```
    //The application ask for the next element, provided by xbe32_nextElement

    char name_buffer[1024];
    char * elem_name = xbe32_getName( item );
    if (elem_name == NULL) {
int elem_id = xbe32_getId(item);
elem_name = xbe32_dictionary_lookup(dictionary, elem_id);
//The element name is look up in the dictionary

if (elem_name == NULL) {
   sprintf(name_buffer, "0x%.8x", elem_id);
   elem_name =  name_buffer;
   }
//In case the element name is not present in the dictionary,
the id is introduced in the XML label instead
    }

    if (!xbe32_isComplex( item )) {
int length,j;
        printf("<%s>", elem_name);
uint16_t * content1 = NULL;
uint32_t * content2 = NULL;
char chain[255];
length = xbe32_getNumValues( item );

    itemType = xbe32_getValuesType( item );

   switch (itemType){
   case STRING_TYPE:
printf("%s",xbe32_getValues( item ));
   break;
   case FLOAT32_TYPE:
     printf(" %f ",*(float *) xbe32_getValues( item ));
   break;
   case FLOAT64_TYPE:
     printf(" %lf ",*(double *) xbe32_getValues( item ));
   break;
   case INT16_TYPE:
```

```
    content1 = (uint16_t *) xbe32_getValues( item );
    for (j = 0; j<length/2;j++){
      printf(" %d", content1[j]);
    }
   break;
   case INT32_TYPE:
    content2 = (uint32_t *) xbe32_getValues( item );
    for (j = 0; j<(length/4);j++) {
      printf(" %d", content2[j]);
    }
   break;
   case INT64_TYPE:
             printf(" %d ",(uint64_t *) xbe32_getValues( item ));
     break;
   case OPAQUE16_TYPE:
      vals = xbe32_getValues(item);
      for (i=0; i<16; i++) {
        printf("%.2x", vals[i]);
      }
   break;
   case OPAQUE4_TYPE:
      vals = xbe32_getValues(item);
      for (i=0; i<length/4; i++) {
        printf(" 0x%.8x", ntohl(vals[i*4]));
      }
    break;
   case OPAQUE2_TYPE:
      vals = xbe32_getValues(item);
      for (i=0; i<length/2; i++) {
        printf(" 0x%.4x", ntohs(vals[i*2]));
      }
    }
  //This case transform the hexadecimal into an element value

    printf("</%s>",elem_name);
  //The element name is printed inside the XML label
    }
    else{ // isExtensible()
```

```
      if (closed == true) {
       printf("</%s>\n", elem_name);
     //The name corresponding the XBE32 element is put inside
the closing label
} else{
     printf("<%s>", elem_name);
     //The name corresponding the XBE32 element is put inside
the opening label
     }
    }

    closed = false;
} while(item != NULL);

  return 0;
}
```

Following the XML got by the application[1]:

```
<service>
 <id>8e9d7823d5ac497c91d0fb07ea0c3fb2</id>
 <servicestate>
  <metainfo>
   <statetimestamp>f85444f4eb</statetimestamp>
  </metainfo>
  <selectstate>
   <0x00002e12> 1.100000 </0x00002e12>
  </selectstate>
 </servicestate>
 <servicemaininfo>
  <servicetype>
   <0x00002112>printer</0x00002112>
  </servicetype>
  <0x00002148>Alice's printer</0x00002148>
  <selectinfo>
   <0x00002845> 0x0000</0x00002845>
   <0x00002d45>14</0x00002d45>
```

[1]The resultant XML is not exactly the same as the first one presented in the previous section since not all the items were present in the dictionary

```
  </selectinfo>
  <color>false</color>
  <duplex>true</duplex>
 </servicemaininfo>
 <servicelocation>
  <inet>
   <ipv4addrs> 0x8b000000</ipv4addrs>
   <ipv6addrs>fe800000000000000202b3fffe3cda7a</ipv6addrs>
  </inet>
  <protocol>
   <0x00002178>ipp</0x00002178>
   <0x00002c77> 0x2f000000 0x2c000000 0x74000000 0x33000000</0x00002c77>
  </protocol>
  <protocol>
   <0x00002178>lpr</0x00002178>
   <0x00002c77> 0x2f000000 0x2c000000 0x74000000 0x31000000</0x00002c77>
  </protocol>
 </servicelocation>
 <serviceaddinfo>
  <0x00002189>Acme Laser Printer 2000</0x00002189>
  <0x00002177>http://www.acme.com/printers/lp200.html</0x00002177>
 </serviceaddinfo>
</service>
```

Chapter 5

CONCLUSIONS AND WORKS FOR THE FUTURE

Este trabajo ha representado dos puntos muy positivos para sus autores. En primer lugar, el hecho de haber implementado la librería en lenguaje C (no hay que olvidar que ya ha sido implementada en Java) hace que el protocolo que representa sea más universal, ya que este lenguaje está caracterizado por su popularidad, diversidad de plataformas, y lo que es más importante, su amplia utilización en la comunidad de software libre. Además de esto, está lo que podría entenderse como la inmediata aplicación de la librería, que es el protocolo XSDF. Este trabajo permite que dicho protocolo de descubrimiento de servicios tenga ya a su disposición una librería XBE32 de implementación C, lo cual hace que su futura implementación en este lenguaje sea más sencilla, ya que todo lo relacionado con la codificación de mensajes queda reducido a la llamada de una serie de funciones.

Además de esto, como se ha mencionado anteriormente, XBE32 no tiene como única finalidad su utilización por parte de XSDF, sino que ha sido diseñado para ser utilizado por cualquier aplicación que necesite de una codificación ligera para utilizar de cara a la red.

Del objetivo principal impuesto, la implementación del contenido del Draft[1] XBE32, podemos decir que se ha conseguido realizar de forma exitosa contando con las siguientes características:

- Flexibilidad: La implementación soporta todas las características de XBE32 incluyendo los elementos extensibles y a las TLV de longitud indetermi-

nada.

- Transparencia para el usuario: Este no tiene que tener en cuenta elementos tan básicos como la TLV, sino que sólo ha de preocuparse por generar elementos de determinado tipo. Esto además se hace de forma sencilla mediante invocaciones de funciones.
- Ligereza: Debido a las características del lenguaje, esta implementación además supone un avance en cuestión de eficiencia, ya que en este caso, la aplicación será más ligera en lo que a su ejecución se refiere consumiendo menos recursos. Esto en determinados entornos puede resultar muy positivo y ventajoso.
- Capacidad para adaptarse a futuros cambios: Dado que la aplicación está claramente dividida en dos capas y las tareas claramente delimitadas por funciones, cualquier ampliación o cambio se vivirá como algo relativamente cómodo y sencillo, y una cantidad de código mínima se verá afectado por los cambios en una determinada tarea.

El proceso de creación ha estado caracterizado por un estudio exhaustivo del Draft[1] que describe XBE32. Este estudio ha sido un punto clave en la implementación, ya que XBE32 es una especificación algo compleja y, aunque en principio la autora subestimó la dificultad que el documento entraña, finalmente después de varios problemas surgidos durante la fase de implementación, se decidió dejar esta algo de lado hasta que el Draft[1] estuviera completamente interiorizado por su parte.

Al margen de los problemas derivados del análisis de la especificación, otra parte compleja, aunque bastante más interesante, ha sido el diseño de la librería. El diseño además de la autora ha contado con la activa colaboración del autor del Draft[1], Manuel Ureña Pascual. Desde el punto de vista de la primera, esta colaboración ha resultado muy estimulante y positiva, ya que, a pesar de que una vez familiarizada con el problema las ideas respecto a su posible materialización fluían de forma bastante concreta, el autor del Draft[1], ha hecho posible que todos esos algoritmos se distribuyeran de una forma lógica, sencilla y elegante entre las dos capas que forman la librería.

A pesar de que las dos fases anteriormente mencionadas han sido sin ninguna duda las más significativas del proyecto, en este apartado del escrito no sería justo no mencionar lo que también ha sido fundamental en este trabajo, la documentación. La documentación ha supuesto para la autora otra fuente de

enseñanza, ya que al margen de cumplir con su función obvia, que es la posibilidad de facilitar al usuario la utilización de la librería, y facilitar posibles cambios y ampliaciones, ha incluido otro reto: aprender las técnicas y dominar las herramientas necesarias para cumplir con los estándares de la comunidad de software libre para la documentación de proyectos. La forma escogida para cumplir este requisito ha sido la de páginas de manual (las conocidas man). A pesar de que han sido relativamente fáciles de editar y generar, han sido un trabajo engorroso debido al particular formato de estas páginas y a la propia estructuración del contenido. Puesto que XBE32 no es un lenguaje de codificación conocido, para facilitar la tarea al usuario final, se ha decidido agrupar las funciones según sus funcionalidades para evitar que dicho usuario tenga que recordar en todo momento los nombres de todas las funciones presentes en la librería y le sea relativamente fácil encontrar la función que cumple con determinada tarea.

Una de las dificultades del proyecto en este caso no ha sido el tiempo como viene siendo habitual con los proyectantes noveles, sino la materia. En principio este proyecto estaba encaminado a ser la implementación de la primera capa del framework XSDF. Dado que en lenguaje C no existía ningún tipo de implementación de XBE32, la parte más básica del proyecto sería realizar una implementación parcial de dicho sistema de codificación para que pudiera ser utilizado por XSDF. A medida que se fue avanzándo en la implementación, quedó patente que si el proyecto debía abarcar hasta la primera capa de XSDF como estaba previsto, el resultado de ambas partes se vería perjudicado en cuanto a calidad se refiere. En el caso de que la implementación de XBE32 estuviese enfocada unicamente a esa primera capa, otras aplicaciones no podrían valerse de sus servicios sin tener que modificar esta o la propia aplicación. Así mismo, esa primera capa de XSDF se verá modificada (u obsoleta) una vez que saliera la implementación total de XBE32, lo cual dadas las circunstancias, sería muy posible. Así que en un momento determinado, cuando la implementación de la primera capa de XBE32 se hubo terminado, fue necesario decidir si se procedía con lo que habría resultado un trabajo incompleto con la implementación de XSDF, o bien se hacía una librería completa de la especificación XBE32.

Todo lo anterior demuestra que a pesar de que este proyecto ha hecho que la autora haya mejorado notablemente en el plano de la programación en el lenguaje de la implementación, también ha vuelto a demostrar que lo que realmente hace que un ingeniero pueda evolucionar como tal, es el trabajo en proyectos de cierta embergadura, ya sea en cuestión de tamaño o complejidad, y que por

tanto hay que seguir trabajando.

Finalmente, para terminar con la lista de objetivos planteados en el primer capítulo, hay que confesar, que algunas de las herramientas que se han utilizado para realizar este trabajo, como el control de versiones, tan sólo se han utilizado en los momentos más tempranos y debido al tiempo que implicaba su utilización, se ha optado por métodos más primitivos (p.e: almacenaje por fechas en correo electrónico y discos duros externos).

Una vez que se han comentado los puntos principales del proyecto, sus dificultades y bonanzas, sólo queda exponer los trabajos futuros.

5.1 Trabajos futuros

Algunos de los trabajos que quedan para el futuro respecto a XBE32, son aspectos de implementación como el comportamiento de la librería cuando para una TLV simple se excede cierto límite de bytes (en la actualidad ese tope está en 65.532 bytes), o el tratamiento de errores de la librería (actualmente, todo tipo de decisiones de comportamiento frente a errores se dejan a la capa superior). Pero sin duda, el trabajo más relevante para el futuro es la implementación completa de XSDF.

XSDF es una evolución del protocolo de localización de servicios (SLP). En todo momento se intenta que esta extensión cumpla con los requisitos definidos por el grupo de trabajo Rserpool (Reliable Server Pooling). Sus principales características son:

- Modelo de servicio mejorado.
- Localización a través de Internet.
- Balanceo de carga.

XSDF, es, como se ha mencionado anteriormente en este trabajo, un framework compuesto por varias capas. Dado que una de ellas, la primera, era el objetivo inicial de este trabajo, creemos que es fundamental presentar tanto esta como las demás, ya que, como se ha mencionado en este mismo capítulo, forman parte de los diferentes trabajos que pueden derivar del que actualmente se está presentando. Estas capas son:

- XSLP (eXtensible Service Location Protocol): Tiene como función proporcionar al usuario información sobre la disponibilidad de determinados servicios presentes en la red del mismo.
- XSRP (eXtensible Service Register Protocol): Se encarga de que los servidores registren la información de los servicios que proporcionan en un directorio centralizado.
- XSSP (eXtensible Service Subscription Protocol): Mediante este protocolo los agentes XSDF pueden suscribirse a información de servicio, de forma que siempre están informados de los servicios disponibles y sus posibles cambios.
- XSTP (eXtensible Service Transfer Protocol): Este protocolo permite distribuir el directorio de servicios entre varias máquinas sincronizadas entre sí.

Appendix A

Installation

This chapter explain how the user should proceed to compile and install the library, the files needed to use the dictionary utility and the man pages.

Following, the steps needed to compile, copy and use the library:

1. To compile the multiple files the library is composed of. To do that, we use the gcc sentence with the -c option:

   ```
   $ gcc -c xbe32_tlv.c xbe32.c xbe32_dictionary.c
   ```

2. After that, we create the library with the ar command:

   ```
   $ ar rs xbe32.a xbe32_tlv.o xbe32.o xbe32_dictionary.o
   ```

3. To create an index inside the library, we execute the next command:

   ```
   $ ranlib xbe32.a
   ```

4. If we desire to copy the library to any part of our system (linux/Unix) we use the option -p with the command cp.

   ```
   $ cp -p xbe32.a directory/
   ```

5. To use the library it is necessary to use gcc with the -L. option. Next, we are going to illustrate how to compile a programm "foo" with our library:

```
$ gcc -o foo -L. -xbe32 foo.o
```

As in any other project, the documentation is necessary for the user to learn the employ of the product. In this case, a very complete example has been given in section 4.2 and , so it is easier for the user to grasp the functioning of the library intuitively. Anyway this library provides a set of manpages to orient the user in the employ of the library.

As it happens with the dictionary utility, there are some requisites needed to have the pages available. This requisites are the only installation needs for a Linux/Unix computer.

To have properly installed the manpages, these need to be stored in the next directories:

- /usr/share/man
- /usr/local/man

Next in this chapter we are going to explain the steps to follow in order to install the dictionary component.

To use the dictionary, it is necessary to create and install it. The instruction to do it are attached bellow:

1. Create a ".txt" file, and open it.
2. Arrange type name and value type in two columns. The first of them must be type name, and the second the value of this type.
3. Repeat step 2) for the each element type.
4. Save the file.
5. To load the dictionary through the library, use the "xbe32_dictionary_loadDictionary" [5].

Following, an example of how must be distributed the dictionary:

[5] See the example in the previous chapter

```
0x0100 service
0x0110 servicestate
0x0111 metainfo
0x0121 servicetype
```

Appendix B

man pages

This appendix shows the man pages corresponding to the library.

```
xbe32(3)        LIBRARY FUNCTIONS       xbe32(3)


NAME
       The functions relative to the elements available for this library are:

       xbe32_createParser

       xbe32_createWriter

       xbe32_destroyParser

       xbe32_destroyWriter

       xbe32_flush

       xbe32_getFlags

       xbe32_getId

       xbe32_getName
```

```
       xbe32_getNumValues

       xbe32_getValues

       xbe32_nextElement

       xbe32_setParserBuffer

       xbe32_setWriterBuffer

AUTHOR
       Lia Bailan <100011513 at alumnos dot uc3m dot es>
```

xbe32_createParser(3) LIBRARY FUNCTIONS xbe32_createParser(3)

NAME

xbe32_createParser, xbe32_createWriter, xbe32_destroyParser, xbe32_destroyWriter - Create/Delete xbe32_parser_t/xbe32_writer_t structures allocating/deallocating the necessary memory.

SYNOPSYS

```
#include xbe32.h

xbe32_parser_t * xbe32_createParser( unsigned char * buf, int len )

xbe32_writer_t * xbe32_createWriter( unsigned char * buf, int len )

void xbe32_destroyParser( xbe32_parser_t * parser)

void xbe32_destroyWriter( xbe32_writer_t * writer )
```

DESCRIPTION

xbe32_createParser Creates a an opaque xbe32_parser_t structure allocating the necessary memory.

The parameters for this function are the buf variable of char* type which represents the buffer which is going to be read, and an int variable type which represents the length of that buffer

xbe32_createWriter Creates an opaque xbe32_writer_t structure allocating the necessary memory.

The parameters for this function are the buf variable, a char* variable which represents the buffer which is going to be read, and an int variable type, len, which represents the length of that buffer

xbe32_destroyParser Free the memory corresponding to the parser structure in the upper layer.

The parameter needed for calling this function is the xbe32_parser_t type variable representing the parser (the parser for the upper layer.

```
       Processing of elements, not TLV).

       xbe32_destroyWriter Free the memory corresponding to the parser
struc
       ture in the upper layer.the upper layer).

       The  parameter  needed  for calling this function is the xbe32_writer_t
       type variable representing the writer (the writer for the upper
layer.
       Processing of elements, not TLV).

AUTHOR
       Lia Bailan <100011513 at alumnos dot uc3m dot es>
SEE ALSO
       xbe32_setParserBuffer, xbe32_setWriterBuffer
```

NAME

xbe32_getFlags - returns an integer with the value of the meta fields "c" continue, and "e" notify error.

xbe32_getId - Returns the the identifier of a given element.

xbe32_getName - Returns the name of an element

xbe32_getNumValues - returns an integer with the number of values inside an attribute element (leaf)

xbe32_getValues - Returns a void pointer to the values of the element.

xbe32_getValuesType - Returns an integer with the type of the values in one attribute element.

SYNOPSYS

```
#include xbe32.h

int xbe32_getFlags ( xbe32_elem_t * elem )

int xbe32_getId ( xbe32_elem_t * elem )

char * xbe32_getName ( xbe32_elem_t * elem )

int xbe32_getNumValues ( xbe32_elem_t * elem )

void * xbe32_getValues ( xbe32_elem_t * elem )

int xbe32_getFlags ( xbe32_elem_t * elem )
```

DESCRIPTION

xbe32_getFlags returns an integer with the value of the meta fields "c" continue, and "e" notify error. With this information, the user knows that an error has happened and what to do (if to continue or not with the processing).

If "c" value is 0, discard this mandatory TLV and stop processing TLVs left

If "c" value is 1, skip this optional TLV and continue processing next TLV

if "e" value is 0, do not report to the sender that this type is unknown

if "e" value is 1, report to the sender that this type is unknown

xbe32_getFlags gets as parameter an xbe32_elem_t type variable.

xbe32_getId returns the identifier of a given element. An element, always have an identifier, and can also have a name.

xbe32_getId gets as parameter an xbe32_elem_t type variable.

xbe32_getName returns the name of an element. To do this, it checks the id of the element against a dictionary utility and finds out if that element has a name. If it has, xbe32_getName returns it.

xbe32_getName gets as parameter an xbe32_elem_t type variable.

xbe32_getNumValues returns an integer with the number of values inside an attribute element (leaf). This function is applicable only to this kind of element (leaf), since they are the only ones capable of carrying data values.

xbe32_getNumValues gets as parameter an xbe32_elem_t type variable.

xbe32_getValues returns a pointer (void type) to the values of the element. This function, as xbe32_getNumValues and xbe32_getValuesType, is just applicable to attribute elements since are the only ones which carry real values.

xbe32_getValues gets as parameter an xbe32_elem_t type variable.

xbe32_getValuesType returns an integer with the type of the values in one element. This function is applicable only to this kind of element (leaf), since they are the only ones capable of carrying data values.

xbe32_getValuesType gets as parameter an xbe32_elem_t type variable.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

xbe32_nextElement(3) LIBRARY FUNCTIONS xbe32_nextElement(3)

NAME

xbe32_nextElement - Returns an xbe32_elem_t structure with the contents of the currently processed element

SYNOPSYS

#include xbe32.h

xbe32_elem_t * xbe32_nextElement(xbe32_parser_t * parser,bool * closedElem)

DESCRIPTION

xbe32_nextElement returns an xbe32_elem_t structure with the contents of the currently processed element. As with its lower layer equivalent, xbe32_tlv_nextTLV, this function returns an element with its contents (data values) in case the processed element is an attribute element (a leaf on the hierarchical tree), and a header with its type, id, and name (if it has one), in case it is a complex element. It must be said, that if it is an attribute extensible element, the name/id of the ele ment will be extracted from a different TLV that the one containing the values. The parameters needed to call this function are an xbe32_parser_t variable which will be representing to the parser of the upper layer application, and a bool type variable to signal that a com plex element has been closed.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

NAME

xbe32_setParserBuffer, xbe32_setWriterBuffer, xbe32_flush - Changes an xbe32_writer_t/xbe32_parser_t buffer for another.

SYNOPSYS

#include xbe32.h

void xbe32_setParserBuffer (xbe32_parser_t * parser, unsigned char * buf, int len)

void xbe32_setWriterBuffer (xbe32_writer_t * writer, unsigned char * buf, int len)

void xbe32_flush(xbe32_writer_t * writer)

DESCRIPTION

xbe32_setParserBuffer changes an xbe32_parser_t buffer for another. This happen when the buffer that the application is reading for pro cessing the elements has reached its end, and the application needs to continue reading from another one.

The parameters for this function are the xbe32_parser_t type variable which represent the upper layer parser structure, the buf variable of char* type which represents the buffer which is going to be read, and an int variable type which represents the length of that buffer

xbe32_setWriterBuffer changes an xbe32_writer_t buffer for another. This happen when the buffer that the application is reading for pro cessing the elements has reached its end, and the application needs to continue reading from another one.

The parameters for this function are the xbe32_writer_t type variable which represented the upper layer structure writer, the buf variable, a char* variable which represents the buffer which is going to be read, and an int variable type, len, which represents the length of that

buffer

xbe32_flush dumps the contents of the buffer which is being written. This is done in order to start the writing in another buffer, or just to dump all the data of the application that is currently in the buffer.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

SEE ALSO

xbe32_createParser, xbe32_createWriter, xbe32_destroyParser, xbe32_destroyWriter

xbe32_tlv_closeTLV(3) LIBRARY FUNCTIONS xbe32_tlv_closeTLV(3)

NAME

xbe32_tlv_closeTLV - Closes a complex TLV.

xbe32_tlv_flush - Dump the contents of the buffer which is being writ ten.

xbe32_tlv_openTLV - Creates the header of a TLV

SYNOPSYS

#include xbe32_tlv.h

int xbe32_tlv_closeTLV(xbe32_tlv_writer_t * writer)

int xbe32_tlv_flush (xbe32_tlv_writer_t * writer)

int xbe32_tlv_openTLV (xbe32_tlv_writer_t * writer, uint16_t type)

DESCRIPTION

xbe32_tlv_closeTLV closes a complex TLV. To do this, checks the TLV pending of being closed (on the stack or through the structure member

num_end_of_tlv) to be sure there is still one in the open state, and after that, writes the corresponding length into the length field (Until this moment, since the TLV which was being processed was complex and the amount of TLV which where going to be nested in it was indefi nite, the length remained unknown).

The parameter needed to call this function is an xbe32_tlv_writer_t variable which will be representing to the writer.

xbe32_tlv_flush Dump the contents of the buffer which is being written. This is done in order to start the writing in another buffer, or just to dump all the data of the application which is currently in the buffer.

Phisically, the only task this function performs is the saving of the number of TLVs open in the member num_end_of_tlv, until the moment
xbe32_tlv_flush is invoked.

The parameter needed to call this function is an xbe32_tlv_writer_t variable which will be representing to the writer.

xbe32_tlv_openTLV Creates the header of a TLV. Introduces the type and the length (if this is known), and takeS the pointer forward to write the next field of the TLV.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

NAME

xbe32_tlv_contents - prints the contents of the currently processed TLV, in case it is simple.

SYNOPSYS

```
#include xbe32_tlv.h

void xbe32_tlv_contents(xbe32_tlv_t * tlv )
```

DESCRIPTION

xbe32_tlv_contents prints the contents of the currently processed TLV, in case it is simple. This function is not necessary to the library, just helps debbuging possible errors.

The parameter needed to call this function is an xbe32_tlv_t type vari able which will represent the TLV which is being processed.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

xbe32_tlv_createParser(3) LIBRARY FUNCTIONS xbe32_tlv_createParser(3)

NAME

xbe32_tlv_createParser - Creates an xbe32 parser structure to process the TLV which are about to be read

xbe32_tlv_paddedLength - Allocates memory for the writer structure

xbe32_tlv_destroyParser - Free the memory used to store an xbe32_tlv_parser_t structure

xbe32_tlv_destroyWriter - Deallocates the memory corresponding to the writer structure

SYNOPSYS

#include xbe32_tlv.h

xbe32_tlv_createParser(unsigned char * buf, int len)

xbe32_tlv_writer_t * xbe32_tlv_createWriter (unsigned char * buf, int len)

xbe32_tlv_destroyParser(xbe32_tlv_parser_t * parser)

void xbe32_tlv_destroyWriter (xbe32_tlv_writer_t * writer)

DESCRIPTION

xbe32_tlv_createParser Creates an xbe32 parser structure to handle the TLV reading process. Allocates memory for the structure and initializes it. This function is responsible of initializing the parser structure. This structure contains several fields:

buffer_start: points to the first byte where the current processing starts.

buffer_end: points to the last byte that can be read in the current stage of processing, that is the start byte plus the total length of

the available bytes of the file.

parser_ptr: points to the current byte which is been processing, so it is initialized pointing to the first byte of the file.

open_tlvs: it counts and stores the total number of tlvs which are currently opened, so it is initialized to NULL, since at first there is no open TLV pending.

error_msg: contains the error string that must be displayed at the moment. Initialize to NULL.

error_code: contains the number of the error that must be spread up to the main application. Initialize to NO_ERROR_ERRCODE.

As income, xbe32_tlv_createParser has the buf variable, which maps the file the library is writting over, and the len variable, which repre
sents the available length for the application.

At the end of the function, the returned value is the parser structure initialized.

xbe32_tlv_createWriter This function initializes the writer structure. It has as income buf, variable which points to the first byte avaiable for the writer, and len, which is the amount of bytes available for the writer.

buffer_start: points to the first byte where the encoding should be writted.

buffer_end: points to the last byte that can be written in the current stage of processing, that is the start byte plus the total length of the available bytes of the file.

buffer_ptr: points to the current byte which is about to be written, so it is initialized pointing to the first byte of the file.

num_end_of_tlv: it is initialized to 0, since at the beggining there is nothing written.

open_tlvs: it counts and stores the total number of TLVs which are currently opened, so it is initialized to NULL, since at first there is no open TLV pending.

error_msg: contains the error string that must be displayed at the moment. Initialize to NULL.

error_code: contains the number of the error that must be spread up to the main application. Initialize to NO_ERROR_ERRCODE.

As income, it has the buf variable, which maps the file the library is writting over,and the len variable, which represents the available length for the application.

At the end of the function, the returned value is the writer structure initialized.

xbe32_tlv_destroyParser free the memory used to store an xbe32_tlv_parser_t structure (a parser varaible).

The parameter for this function is the xbe32_tlv_parser_t type variable representing the parser.

xbe32_tlv_createWriter deallocates the memory corresponding to the
writer structure.

The parameter for this function is the xbe32_tlv_writer_t type variable representing the writer.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

NAME

xbe32_tlv_getContinueFlag - returns a boolean value signalling the possibility of continuing the current parsing operation after find out that an error in the structure of the TLV has ocurred

xbe32_tlv_getErrorFlag - returns a boolean value that indicates if an error that has been found in the structure of the currently processed TLV should be reported to the upper application

xbe32_tlv_getLength - Returns the length of the TLV which is being currently handled

xbe32_tlv_getMeta - Returns the basic type of a defined one.

xbe32_tlv_getNumValues- Returns the number of values inside a TLV

xbe32_tlv_getType - Returns the type of the TLV which is being currently handled

xbe32_tlv_getValues - returns a char pointer that points to the first of the bytes corresponding to the values in the single TLV which is being processed

xbe32_tlv_getValuesLength - Returns the length of the payload without the padding

SYNOPSYS

```
#include xbe32_tlv.h

bool xbe32_tlv_getContinueFlag ( uint16_t type )

bool xbe32_tlv_getErrorFlag ( uint16_t type )

uint16_t xbe32_tlv_getLength ( xbe32_tlv_t * tlv )
```

```
uint16_t xbe32_tlv_getMeta ( int type )

int xbe32_tlv_getNumValues ( xbe32_tlv_t * tlv )

uint16_t xbe32_tlv_getType ( xbe32_tlv_t * tlv )

unsigned char * xbe32_tlv_getValues ( xbe32_tlv_t * tlv )

xbe32_tlv_getValuesLength ( int length )
```

DESCRIPTION

xbe32_tlv_getContinueFlag returns a boolean value signalling the possibility of continuing the current parsing operation after find out that an error in the structure of the TLV has ocurred. The bit signalling this event is present in the meta section of the type field.

The parameter needed to call this function is an uint16_t type variable which will represent the type of the TLV.

xbe32_tlv_getErrorFlag returns a boolean value that indicates if an error that has been found in the structure of the currently processed TLV should be reported to the upper application

The parameter needed to call this function is an uint16_t type variable which will represent the type of the TLV.

xbe32_tlv_getLength returns the length of the TLV which is being currently handled. This length is returned in hexadecimal notation and represent the total length of the TLV including the header but not the padding. During the process the function takes care of the possible problems with the LITTLE ENDIAN notation.

The parameter needed to call this function is an int type variable which will represent the length of the TLV.

xbe32_tlv_getMeta Returns the basic type of a defined one. This type is

the result of substract the meta characters of the complete type. Those meta characters indicate what the actions should be taken in case of error during the processing of the TLV.

xbe32_tlv_getNumValues returns the number of values inside a TLV. This values must be simple values, this function is not applicable to com plex TLV.

The parameter needed to call this function is an xbe32_tlv_t type vari able which will represent the TLV which is being processed.

xbe32_tlv_getType Returns the type of the TLV which is being currently handled. This type is returned in hexadecimal notation. During the pro cess the function takes care of the possible problems with the

LITTLE

ENDIAN notation.

The parameter needed to call this function is an xbe32_tlv_t type vari able which will represent the TLV which is being processed.

xbe32_tlv_getValues returns a char pointer that points to the first of the bytes corresponding to the values in the single TLV which is being processed. The values are stored in binary form.

xbe32_tlv_getValuesLength Returns the length of the payload without the padding.

The parameter needed to call this function is an int type variable which will represent the total length of the TLV (header + payload - padding).

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

SEE ALSO

xbe32_tlv_getErrorFlag, xbe32_getFlags

xbe32_tlv_isComplex(3) LIBRARY FUNCTIONS xbe32_tlv_isComplex(3)

NAME

xbe32_tlv_isComplex - Returns a boolean value that tells if the TLV processed is complex or not

SYNOPSYS

#include xbe32_tlv.h

bool xbe32_tlv_isComplex (uint16_t type)

DESCRIPTION

xbe32_tlv_isComplex returns a boolean value that tells if the TLV pro cessed is complex or not.

The parameter needed to call this function is an int type variable which will represent the type of the TLV.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

NAME

xbe32_tlv_nextTLV - Returns a TLV structure with the contents of the currently processed TLV

SYNOPSYS

#include xbe32_tlv.h

xbe32_tlv_t * xbe32_tlv_nextTLV(xbe32_tlv_parser_t * parser, bool * closed)

DESCRIPTION

xbe32_tlv_nextTLV is in charge of process every TLV on the buffer. For each one, it returns a TLV structure with the values corresponding to the TLV in case this is a simple one, or just the header in case it is a complex one. In addition to that, it returns a boolean value that tells if the last complex TLV (in case there is one) was closing with the last TLV processed. While xbe32_tlv_nextTLV processes the TLVs, the function advances the main pointer of the parser structure, the member

parser_ptr , to point the next TLV to process (that is, the length of the TLV which is being processed).

The parameters needed to call this function are an xbe32_tlv_writer_t variable which will be representing to the writer, and a bool type variable to signal that a complex TLV has been closed.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

xbe32_tlv_setParserBuffer(3) LIBRARY FUNCTIONS xbe32_tlv_setParserBuffer(3)

NAME

xbe32_tlv_setParserBuffer - Changes the buffer for an xbe32_tlv_parser_t parser structure

xbe32_tlv_setWriterBuffer - Changes the buffer for an xbe32_tlv_writer_t writer structure

SYNOPSYS

#include xbe32_tlv.h

xbe32_tlv_setParserBuffer (xbe32_tlv_parser_t * parser, unsigned char * buf, int len)

void xbe32_tlv_setWriterBuffer (xbe32_tlv_writer_t * writer, unsigned char * buf, int len)

DESCRIPTION

xbe32_tlv_setParserBuffer changes the buffer for an xbe32_tlv_parser_t parser structure. It is used in the case the read buffer has reached its end and the processing has not been finished, so another buffer is waiting to be read. To do this, it is necessary to apply some changes over the parser structure: member buffer_start must point to the start of the new buffer; buffer_end must point to the end of the mentioned new buffer; and the member parser_ptr must point to the start of the new buffer as buffer_start (the difference between the latter and parser_ptr, is that parser_ptr will change its position along the pro cessing).

The parameters introduced in this function are: a variable of xbe32_tlv_parser_t type which is representing the parser structure; a char* pointer to the buffer which must be read; and an int variable with the length of the new buffer to be read.

changes the buffer for an xbe32_tlv_writer_t writer structure. It is

used in the case the read buffer has reached its end and the processing has not been finished, so another buffer must be read (the data has not finished, but the buffer did) in order to encode all the message. To do this, it is necessary to apply some changes over the writer structure: member buffer_start must point to the start of the new buffer; buffer_end must point to the end of the mentioned new buffer; and the member buffer_ptr must point to the start of the new buffer as buffer_start (the difference between the latter and buffer_ptr, is that bufferr_ptr will change its position along the processing).

The parameters introduced in this function are: a variable of xbe32_tlv_writer_t type which is representing the writer structure; a char* pointer to the buffer which must be read; and an int variable with the length of the new buffer to be read.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

xbe32_tlv_writeTLV(3) LIBRARY FUNCTIONS xbe32_tlv_writeTLV(3)

NAME

xbe32_tlv_writeTLV - Writes the content of a simple TLV.

SYNOPSYS

#include xbe32_tlv.h

int xbe32_tlv_openTLV (xbe32_tlv_writer_t * writer, uint16_t type)

DESCRIPTION

xbe32_tlv_writeTLV writes the content of a simple TLV. Writes the pay load of a simple TLV including the padding. In addition to that, takes the pointer forward as far as necessary to write the next TLV (that is, the length of the current TLV).

The parameters needed to call this function are an xbe32_tlv_t type variable which will represent the TLV which is being processed and an int type variable which will represent the type the TLV.

AUTHOR

Lia Bailan <100011513 at alumnos dot uc3m dot es>

version 0.1 March 2009 xbe32_tlv_writeTLV(3)

Bibliography

[1] Uruena, M. y Larrabeiti, D., "eXtensible Binary Encoding", <draft-uruena-xbe32-00>, Marzo 2004.

[2] Uruena, M. y Larrabeiti, D., "Overview of the eXtensible Service Discovery Framework", <draft-uruena-xsdf-overview-00>, Marzo 2000.

[3] Uruena, M. y Larrabeiti, D., "Overview of the eXtensible Service Discovery Framework: Common Elements and Procedures", <draft-uruena-xsdf-common-00.txt>, Marzo 2004.

[4] Uruena, M. y Larrabeiti, D., "eXtensible Service Location Protocol (XSLP)", <draft-uruena-xslp-00.txt>, Marzo 2004.

[5] Uruena, M. y Larrabeiti, D., "eXtensible Service Registration Protocol (XSRP)" <draft-uruena-xsrp-00.txt>, Marzo 2000.

[6] Uruena, M. y Larrabeiti, D., "eXtensible Service Subscription Protocol (XSSP)" <draft-uruena-xssp-00.txt>, Marzo 2004.

[7] ITU-T, "ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encodign Rules (DER)", X.690, Diciembre 1997.

[8] Fernández, M., Diseño e Implementación del API del eXtensible Binary Encoding (XBE32), Marzo 2005.

[9] José Ángel Martínez Usero y Elsa Palacios Ramos "XML: un medio para fomentar la interoperabilidad, explotación y difusión de contenidos en la administración electrónica"

[10] ITU-T Recommendation X.690: SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS OSI networking and system aspects – Abstract Syntax Notation One (ASN.1) Information technology – ASN.1

encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
July 2002

[11] SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS OSI networking and system aspects – Abstract Syntax Notation One (ASN.1) Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)
December 2001