Ember+ Specification

Version: 2.5
Author: Marius Keuck, Philip Boger
Date: 07 – 2012

07/2012	▪ Added a note about the formula format. ▪ Described a rule to detect empty nodes.
06/2012	▪ Updated the Glow Dtd to version 2.5, which introduces a more flexible way to transport streams and qualified nodes and parameters.
05/2012	▪ Changed the numbers of the Subscribe and Unsubscribe command ▪ Updated the Glow Dtd to version 2.4
04/2012	▪ The tags of nodes and parameters are now context specific. ▪ A node can now be marked as root. ▪ Added the EnumMap property and the description of the EnumCollection and EnumEntry types.
02/2012	▪ Removed the tag information in the command list since a command is only represented by its number.
02/2012	▪ Initial draft.

Content

Introduction

The Ember+ protocol has been designed to allow the communication between two endpoints, one being the data provider and the other one being the consumer. The data provider usually is a piece of hardware which offers a set of controllable parameters, while the consumer may be a control- or monitoring-system which provides access to these parameters. This document describes the Ember+ protocol specification and the encoding it uses. Furthermore it provides some implementation hints and gives several code examples that demonstrate the use of this protocol.

The chapter "EmBER" describes the basics of the *Embedded Basic Encoding Rules* (EmBER), which is the encoding used by this protocol. The code samples listed in this document refer to the EmBER library implementation written in C++. The library is freely available and usually published together with this document. It implements the subset of BER which is required for this protocol. BER is a well-known encoding standard. It is platform independent, generates small data packets and offers an XML-like flexibility. It provides encoding rules for all common data types, like integers, decimal numbers, strings, byte arrays and more. Additionally, it provides container types called set and sequence. EmBER uses a TLV (Type, Length, Value) system in order to specify what kind of data is currently present in the encoded data buffer. This allows extending the protocol by user defined data types, similar to XML where a node may have any name but also indicates what kind of attributes or data it contains.

The chapter "Ember+ specification" provides detailed information about the types, properties and commands being used. The Ember+ specification relies on BER and defines a set of objects in ASN.1 notation, which are also listed in this document.
One of the important points Ember+ put a focus on is to keep communication between the peers as generic as possible. That means that the consumer doesn't have to have any kind of detailed knowledge of the device it is connected to. Instead, the device is responsible of providing its structure when the consumer requests it. This allows a consumer to control any kind of device as long as they follow the specification.

Chapter number three, called "Ember+ Usage" describes how a provider represents its data, how a consumer queries the structure and how it can change a parameter value.

In "EmBER Library" the different layers of the C++ library implementation are described. Additionally, the section shows how the library can be used to encode or decode data.

The chapter "Message Framing" describes the framing protocol which is used to transport encoded EmBER packets, called S101. The library also contains a basic encoder and decoder framework that can be used to create valid S101 messages.

"Behaviour rules and guidelines" lists how a consumer and a provider must behave in several situations. Besides, this chapter provides some general guidelines about how a tree should look like.

The Ember+ object definition is called “Glow”, the specification is listed in the chapter “Glow DTD ASN.1 Notation”.

The library and the framework referred to by this document provide everything needed in order to easily implement the Ember+ protocol. It makes it unnecessary to have detailed knowledge about the encoding and decoding algorithms.
Additionally to the C++ library mentioned and used in this document, there is also a variant written in standard C (called libember slim) and a C# version for the .NET Framework 2.0.

EmBER

Introduction

This chapter describes the data types supported in this library and the restrictions.

About BER

BER stands for Basic Encoding Rules. The rules describe how data is being encoded and decoded. The encoding algorithms are fast and produce small data packets, which is a good reason to use this encoding on embedded systems. Each data entity is encoded by a tag, the data length and the value itself, in short terms TLV. The tag simply defines the type, while the length field contains the length of the encoded value in bytes. Both, the tag and the length are encoded as well, which reduces the size of the resulting data packet even more. The structure of a tag is described in the second chapter.

Objectives

EmBER has been developed in order to provide a way to encode data for transfer between different electronic devices with varying hardware resources, independent from the transmission medium and protocol. Ember can also be used to serialize data to non-volatile memory (e.g. a file on a hard disk).
Ember has been designed to provide the following assets:

1. Conformance to a well-known and widely adopted standard
2. Compact size of the encoded data to minimize transmission load
3. Platform independence
4. XML-like flexibility
5. Binary storage of values for fast encoding and decoding

Conformance

Ember forms a subset of the Basic Encoding Rules (BER), an ITU and ISO standard developed by the ASN.1 consortium (ASN.1: ITU-T X.680, ISO/IEC 8824-1; BER: ITU-T X.690, ISO/IEC 8825-1).
ASN.1 and BER are used by widely adopted technologies and standards like:

- LDAP, Active Directory
- PKCS (Public Key Cryptography Standard)
- X.400 Electronic Mail
- Voice Over IP
- SNMP
- UMTS.

It is recommended to read the documents listed in the section “References” in order to be able to fully comprehend this documentation.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

References

This document refers to the following specifications:

- BER (ITU-T X.690 07/2002) http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf
- ASN.1 (ITU-T X.680 07/2002) http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

Additional resources:

- Brief introduction to ASN.1 and BER from the Microsoft Knowledge Base http://support.microsoft.com/kb/252648
- “ASN.1 Complete”, a very comprehensive and freely downloadable book covering ASN.1 and BER by Professor John Larmouth http://www.oss.com/asn1/larmouth.html

Basics

Overview

Ember encoded data must be decodable by any fully featured BER decoder. An Ember decoder on the other hand is not required to support the entire BER standard, since it is only required to implement the BER subset defined in this document. Therefore, an Ember encoder shall apply all the restrictions to the BER encoding defined in this document.
The Paragraph “Length Forms” and following define the most important parameters for the BER encoding that turn Ember into a mere subset of BER.

Tagging [X.690 8.14]

The tagging environment shall be explicit without any exceptions. An encoded value usually contains two tags, the outer- or application-tag, which defines the meaning or semantics of the data, and the inner-tag which indicates the type of the encoded object. Besides the tag, the length of the encoded data is stored. So every object being encoded contains a Tag/Type, a Length, and the Value, in short TLV. An encoding example can be found in the chapter “The EmBER library”.

Length forms [X.690 8.1.3]

1. Generally, both length forms (definite and indefinite) are allowed; usage of the indefinite form though shall be permitted for containers only [X.690 8.1.3.2].
2. The length forms used by the inner and outer tag generated through explicit tagging must be identical. So if the outer tag specifies an indefinite length, the inner tag also has to specify an indefinite length.

Container Usage

The only containers that may be used are “Set” [X.690 8.11] and “Sequence” [X.690 8.10].

Basic document structure

The document must start with a container, and there shall only be one container at the top-level (like in the XML standard).

Types

Overview

The following BER types are supported by Ember:

1. Boolean
2. Integer
3. Real
4. UTF8String
5. Octet String
6. Set
7. Sequence
8. Relative Object Identifier

Boolean [X.690 8.2]

The decoder shall interpret 0 (zero) as “false” and any non-zero number as “true”. It is recommended for the encoder to use FF16 to encode “true”.

Integer [X.690 8.3]

Integers shall have a maximum size of 64 bits; the decoder may choose the data type to hold the decoded integer depending on the actual length. (E.g. 32 bit integer if the unsigned value is < 232, 64 bit otherwise).
Special care must be taken by the encoder to comply to rule [X.690 8.3.2]: the first 9 bits of the encoded integer value must not be the same.
To verify the correctness of the implementation, the following table should be used:

Input Value	**Encoded result**	
Integers (base 10)	*Length octets*	*Value octets (base 16)*
1	01	01
-1	01	FF
255	02	00 FF

127	01	7F
128	02	00 80
-128	01	80
$2^{16}-1$	03	00 FF FF
2^{15}	03	00 80 00
-2^{15}	02	80 00

Real [X.690 8.5]

Real values must be encoded in binary form [X.690 8.5.6], base 2 [X.690 8.5.6.2]. Maximum precision shall be double (64 bit).

UTF8String [X.690 8.21]

Only characters in the ANSI Western range (0 through 255) are supported. This is subject to change in Ember versions to come.

Octet String [X.690 8.7]

This type shall be used to encode junks of binary data. The encoding must be primitive; splitting the octet string [X.690 8.7.3] is not permitted.

Relative Object Identifier [X.690 8.20]

A relative object identifier is a sequence of integers, which identify an object within a tree. The object identifier is encoded by writing the encoded length first, followed by all components encoded as multi-byte integers.

Set [X.690 8.11]

Every item in a “Set” container shall be implicitly optional; therefore an empty set is allowed. The tags of values in a Set container must be unique within the container context [X.680 26.3]. The order of values in a set shall be arbitrary [X.680 26.6].

Sequence [X.690 8.9, X.690 8.10]

This container shall represent the concept of either an ordered collection or a list type (Sequence-of [X.690 8.10]). The order of child values may be semantically significant [X.690 8.10.3]; therefore, tags do not necessarily have to be unique within the container context, though it is recommended to ensure tag uniqueness.

Implementation recommendations

Encoder

Ember puts a complex job upon the encoding entity: the encoder has to calculate the number of octets the encoding process will yield before the actual encoding is done. This is because the length octets are transmitted prior to the data octets. To make things worse, the tag and length octets preceding the data also vary in number. This makes it almost impossible to encode entire containers on the fly.
For encoding entities equipped with powerful hardware resources (like workstation and server PCs), it is therefore recommended to build the encoding in a DOM – like object tree, thereby calculating the length of each container node. This tree is easy to modify and quick to write out, but can possibly consume lots of memory.
For encoding entities that are based on less powerful hardware like embedded devices, hope lies in the generosity of the Basic Encoding Rules, which Ember does not confine:

- To encode containers, the encoder is free to use the indefinite length form instead of pre-calculating the number of value octets (which in turn contain each component's tag, length and value octets).
- The encoder is free to use the definite long length form [X.690 8.1.3.5] even for lengths smaller than 128. This allows fixing the number of octets used for length fields to – as an example – four.
- Depending on the DTD, a fixed number of octets can also be used to encode tags. E.g. if the DTD defines a maximum tag number of 127 (remember that tags are context-specific by default!), an encoder can always use two bytes to encode a tag – with the second byte containing the actual tag number and the five least significant bits of the first byte all set to "1".
- Most primitive types inherently have a maximum length that is dependent on the encoder's hardware. E.g. for a 16 bit embedded system, the maximum number of octets an integer can be encoded into is usually 4 (for 32 bit "long" values). Thus, the value can be encoded into a fixed-length memory block, returning the number of bytes written.

Decoder

Decoding an Ember stream is quite straight-forward and the implementation should be the same for systems with powerful hardware resources as for embedded systems.
A decoder should implement one function to read and store the next item (tag, length and value) from the input stream. Additional functions should be implemented to convert the current value to the type indicated by the inner tag.
Stepping through the containers can either be implemented using recursion or by stacking container objects.

Ember+ specification

Introduction

This chapter describes the Ember+ specification, which is written in ASN.1 and called the Glow DTD (Design Type Document). The DTD defines several object types that are required for a device to represent its structure. The number of types is kept at a minimum so that it can be used for almost any kind of device. The most frequently used types are node and parameter. A node represents a device and its elements, like modules, any kind of extension cards or attached panels. Additionally, it may be used to introduce categories, like “Sources” and “Targets”.
A parameter then contains the properties of a control, like the range, the access (read, write, read-write, none), the name and so on.

Data types and properties

The tag format

As already mentioned, a tag may indicate the meaning and the type of the encoded data that follows next. A tag consists of two components, a class - which uses one byte - and a number, using up to four bytes. The combination of both defines the type.

The tag class accepts the following values:

Name	Value	Description
Universal	0x00	Predefined types. Should never be used.
Application	0x40	Application specific tags have the same meaning wherever seen and used.
Context	0x80	The meaning context specific tags depends on the location where they are seen
Private	0xC0	A special version of the context specific tag

If the tag defines a container type, like set or sequence, the 6th bit of the tag is set. Alternatively a binary OR operation can be performed (Class | 0x20). The meaning of the tag number depends on the class. If it is set to “Universal” it defines one of the default data types. The following table shows a list of standard BER types:

Number	Type
1	Boolean
2	Integer
3	Bitstring
4	Octet String
5	Null
6	Object Identifier
7	Object Descriptor
8	External
9	Real
10	Enumerated
11	Embedded Pdv
12	UTF8 String
13	Relative Object
16	Sequence
17	Set
18	Numeric String
19	Printable String
20	Teletex String
21	Videotex String
22	IA5 String
23	UTC Time
24	Generalized Time
25	Graphic String
26	Visible String
27	General String
28	Universal String
29	Unspecified String
30	BMP String
31	Last Universal
0x80000000	Indicates an application defined type

Note that only the required subset of the universal types is supported in this library.

For example, a tag with the class set to “Universal” and the number set to 1 indicates that the encoded data represents a Boolean value.

Application defined tags

All other tag classes (Application and Context) may be used to define application specific types or, if used as (outer) application-tag, to indicate the usage. Each encoded object contains two tags, the first one identifying the meaning or semantics of the data or object, while the second one specifies the data type.
The Glow specification uses a set of attributes which are used to describe a node or parameter. Two common attributes are the identifier and the description string. The following section lists all attributes used by the Glow DTD. Since these attributes differ for each object type they are context specific. Please note that “Value” simply means that a property may be an Integer, Real,

UTF8String, an octet string value or a Boolean. The details about the “Value” type can be found in the next chapter.

EmBER specific properties

This chapter describes all properties that are used within this protocol.

Application specific properties

Root

Tag: Application – 0

Type: RootElementCollection, StreamCollection (Container)

Each ember message has to start with a RootElementCollection or the StreamCollection.

Default Element

Tag: Context – 0

Type: Usually used for a collection of nodes or parameters, where the tag is not used.

Node properties

All Node properties are context specific.

Number

Tag: Context – 0

Type: Integer

Each node must have a number that is unique within the current scope. It must not change while a session is active because it may be used to identify a node or parameter by using this number instead of its string identifier.

Contents

Tag: Context – 1

Type: Set

This set contains the optional properties of a node.

Children

Tag: Context – 2

Type: Sequence

Contains the child-nodes and parameters of this node.

QualifiedNode properties

All QualifiedNode properties are context specific

Path
Tag: Context – 0
Type: Relative Object Identifier

Contains the numeric path of this node.

Contents
Tag: Context – 1
Type: Set

This set contains the optional properties of a node.

Children
Tag: Context – 2
Type: ElementCollection

Contains the child nodes and parameters of this node.

Node Contents

Identifier
Tag: Context – 0
Type: UTF8String

The identifier of a node. This name must be unique within the current scope, which means that all child nodes of one parent must have different names. The identifier of an entity must not change!

Description
Tag: Context – 1
Type: UTF8String
Display name of a node. This property shall be displayed by a user interface if it is provided. Otherwise, the identifier should be used.

IsRoot
Tag: Context – 3
Type: Boolean

Indicates if the current node is a root node. This flag may be used by providers acting as a kind of proxy, where several sub trees of different devices are merged into one tree.

Parameter properties

All Parameter properties are context specific.

Number

Tag: Context – 0
Type: Integer

Each parameter must have a number that is unique within the current scope. It must not change while a session is active because it may be used to identify a parameter by using this number instead of its string identifier. This number must not change.

Contents

Tag: Context – 1
Type: Set

This set contains the optional properties of a parameter.

Children

Tag: Context – 2
Type: ElementCollection

This collection may contain a command for the current parameter.

QualifiedParameter properties

All Parameter properties are context specific.

Path

Tag: Context – 0
Type: Relative Object Identifier

Contains the numeric path of this parameter.

Contents

Tag: Context – 1
Type: Set

This set contains the optional properties of a parameter.

Children

Tag: Context – 2
Type: ElementCollection

This collection may contain a command for the current parameter.

Parameter Contents

Identifier
Tag: Context – 0
Type: UTF8String

The string identifier of a parameter. This name must be unique within the current scope, which means that all child nodes of one parent must have different names. The identifier of an entity must not change!

Description
Tag: Context – 1
Type: UTF8String

Display name of a parameter. This property shall be displayed by a user interface if it is provided. Otherwise, the identifier should be used.

Value
Tag: Context – 2
Type: Value

The current value of a parameter. Supported types are Integer, Real and String. If the value contains an enumeration the type must be integer and the value contains the index of the enumeration entry to display, starting at index 0.

Minimum
Tag: Context – 3
Type: Value

The smallest value allowed. May be real or integer, but should match the type set in Value.

Maximum
Tag: Context – 4
Type: Value

The largest value allowed. May be real or integer. If the value is of type string it defines the maximum length the string may have.

Access
Tag: Context – 5
Type: Integer

Indicates how this parameter can be accessed. The following values are available:

- None (0)
- Read Only (1)
- Write Only (2)
- Read/Write (3)

Format
Tag: Context – 6
Type: UTF8String

Optional format string. This property may contain a C-Style format string which may be used to append a unit or define the number of digits that should be displayed.

Enumeration
Tag: Context – 7
Type: UTF8String

A single string containing the values of an enumeration, separated by a line feed ('\n', 10). The values must internally be enumerated from 0 to N. If an entry shall not be displayed in a user interface, the entry must begin with '~'.

Factor
Tag: Context – 8
Type: Integer

This property may be used if a device is not able to process decimal values. It may then provide a factor instead. The consumer then has to divide the reported value when it displays it and multiply it with this factor when it wants to change the parameter. The type of the factor is integer.

IsOnline
Tag: Context – 9
Type: Boolean

Online state. When a node reports an offline state, the consumer should assume that all child nodes are offline as well. On the other side, when a node reports an online state, only the parents of this node may be changed to online.

Formula
Tag: Context – 10
Type: UTF8String

Optional mathematical terms. The device must provide two formulas separated by a linefeed character ('\n', 10). The first formula is used to transform the device value into a display value (provider to consumer), while the second one is used transform it back to a device value again (consumer to provider).

Note
Please note that there are two formats supported:

- A term can be provided as RPN term, without a character that represents a variable for the current device value. An example would be 5*, which would multiply the current device value with 5. The device value always has to be pushed on the computation stack first.
- When the expression is enclosed in parentheses a term is interpreted as mathematical expression, including the $ character as placeholder for the current device value. A valid term would be (5 * $).
  A detailed description of all supported functions can be found in the document "Ember+ Formulas".

The RPN variant is provided for backward compatibility.

Step
Tag: Context – 11
Type: Integer

This property is currently not supported.

Default
Tag: Context – 12
Type: Value

Default value of a parameter.

Type
Tag: Context – 13
Type: Integer

Used to provide a hint about the value type of the parameter. The following values are currently available:

- Integer (1)
- Real (2)
- String (3)
- Boolean (4)
- Trigger (5)
- Enum (6)
- Octets (7)

StreamIdentifier
Tag: Context – 14
Type: Integer

A number used to identify an audio level meter. Since peak meter data is usually reported frequently (like every 80ms) it may be necessary to use a separate transport layer like UDP in order to transmit their values. That's why audio streams require a globally unique identifier. Their values are being reported in a separate container, the StreamCollection.

EnumMap

Tag: Context – 15

Type: StringIntegerCollection

This field provides a more complex description of an enumeration. The StringIntegerCollection is a sequence of a StringIntegerPair which is tuple name, value tuple. This field can be used if a device is unable to use a consecutive enumeration.

StreamDescriptor

Tag: Context – 16

Type: StreamDescription

A StreamDescriptor is required when a single stream entry contains more than one value and is provided as octet string. The descriptor then provides the internal data offset and the stream value format. The formats available can be found in the DTD at the end of the document.

Application defined commands

In addition to these properties there is also a small list of commands. Commands are only used by the consumer, the provider always only has to respond or report its current status. The command type is determined by the number of a command, which must always be provided.

GetDirectory

Number: 32

This command may be executed on a node and a parameter. It requests all child nodes and parameters of the node containing this command. The response shall also include all attributes of the reported entities.
The purpose of this command is to obtain the complete or only a part of the structure of a data provider.
If the size of the structure is limited, a data provider may also return its whole structure when it receives a GetDirectory request. This may especially be useful for embedded devices with limited cpu power.
Additionally, this command implicitly indicates that a consumer is interested in the entities below the node the command has been sent for. That means, that parameters belonging to this node should report value changes to the consumer automatically.
When a consumer performs a GetDirectory request on a node without any children, the

node must report itself without its identifier. That way, the consumer knows that a query has been performed on this node and that it has no child elements.

This command can also be applied to a parameter to query its properties.

Besides, a *dirFieldMask* can also be provided, which is tagged with “Context – 1”. This mask determines the properties a consumer is interested in:

- All (-1)
- Default (0)
- Identifier (1)
- Description (2)
- Value (4)

All and *Default* should return the same result, which is a node or parameter with all its properties set. When a consumer requests a property that doesn’t exist, the provider should report the node or parameter without any properties.

Subscribe

Number: 30

This command is used to subscribe to a parameter. Currently, parameters without a stream identifier should report value changes automatically, while all parameters which have a stream identifier must be subscribed in order to receive updates.

Unsubscribe

Number: 31

This command is used to unsubscribe from a node or parameter. When applied to a node, all nodes and parameter owned by that node must be unsubscribed as well.

Application defined types

In many cases the universal types suffice the requirements. Nevertheless it is possible to declare application defined types. Ember+ defines several types like Node, Parameter and Command.
The tag class of user defined types is set to "Application"-specific, while the number determines the type itself. The EmBER library provides a Type class which can be used to distinguish between universal and application defined types. The following section lists the types and their meaning used and defined by the Glow specification.
The complete ASN.1 notation of the Glow object types is listed in the chapter "Ember+ ASN.1 Notation". Additionally, the chapter contains detailed descriptions about all objects for clarity.

Element

The element is a choice that contains one of the following types:

- Parameter
- Node
- Command
- StreamCollection

RootElement

- Element
- QualifiedNode
- QualifiedParameter

The type an element contains can be determined by the type tag. A QualifiedNode and QualifiedParameter may only appear at root level, to reduce the complexity of the tree structure.

Value

A value is a choice that contains one of the following types:

- Integer
- Real
- UTF8String
- Boolean
- Octet String

Parameter

Type Tag: Application – 1
Type: Sequence
Members:

- Number
- Contents, which may contain:
  - Identifier
  - Description

    - Value
    - Minimum
    - Maximum
    - Access
    - Format
    - Enumeration
    - Factor
    - Online
    - Formula
    - Default
    - Command
    - StreamIdentifier
    - EnumMap
    - StreamDescriptor
- Children

A parameter represents an entity that at least contains a value.

When a data provider reports a parameter, the message must always contain the number. All other properties are optional and are listed within a set with Tag Context – 0. In general, automatic change notifications should only contain the properties that have change.

QualifiedParameter

Type Tag: Application – 9
Type: Sequence
Members:

- Path
- Contents
- Children

The contents are equal to the Parameter type. In comparison to the default Parameter object, the QualifiedParameter reports its complete path instead of its number. That way a parameter that is located in a large subtree can be reported without the need to construct the whole tree structure, which saves cpu time and memory.

Command

Type Tag: Application – 2
Type: Sequence
Members: Number

A command may be appended to a node or a parameter as child element inside the ElementCollection property. A command only contains an integer identifying the command type, which may be GetDirectory, Subscribe or Unsubscribe.

Note: When the command is GetDirectory, an additional property called *DirFieldMask* can be provided. For details please see the GetDirectory command.

Node

Type Tag: Application – 3
Type: Sequence
Members:

- Number
- Contents, which may contain:
  - Identifier
  - Description
  - IsRoot
- Children

A node represents a device or one of its components. Like the parameter, is must contain a number which identifies the node while the session is active. All other properties are optional and therefore listed in a separate set.

QualifiedNode

Type Tag: Application – 10
Type: Sequence
Members:

- Path
- Contents
- Children

The contents are equal to the Node type. Compared to the default Node, the QualifiedNode reports its complete path, instead of only providing its number. That way, a node that is located in a big subtree may also be reported at root level.

ElementCollection

Type Tag: Application – 4
Type: Sequence
Members: Entries of type Element

The ElementCollection is a sequence of Element types. A node or a parameter may contain an ElementCollection with commands or child nodes.

RootElementCollection

Type Tag: Application – 11
Type: Sequence
Members: RootElement

Root

App. Tag: Application – 0
Type: Choice of RootElementCollection or StreamCollection

StreamEntry

Type Tag: Application – 6
Type: Sequence
Members: StreamIdentifier, Value

Stream entries are used to report audio level data. Since these values change frequently, they may be transmitted via a different transport layer, like UDP. A StreamEntry is a sequence consisting of the unique stream-identifier and the current value. The StreamEntry contents must be transmitted in the specified order.

StreamCollection

Type Tag: Application – 5
Type: Sequence
Members: Entries of type StreamEntry

A sequence of StreamEntry elements.

StringIntegerPair

Type Tag: Application – 7
Type: Sequence
Members: Name, Value

A tuple containing a enumeration name and an integer value. This type is used by the StringIntegerCollection, which is used by the EnumMap property.

StringIntegerCollection

Type Tag: Application – 8
Type: Sequence
Members: Entries of type StringIntegerPair

A sequence that contains StringIntegerPairs.

Ember+ usage

With the object types defined in the previous chapter it is now possible to represent a piece of hardware, for example a frame controller which has one or two power supplies and ten slots for different kinds of cards. All types mentioned before are available in the Ember library, what makes it intuitive and easy to use. This chapter describes how devices are being represented in Glow.

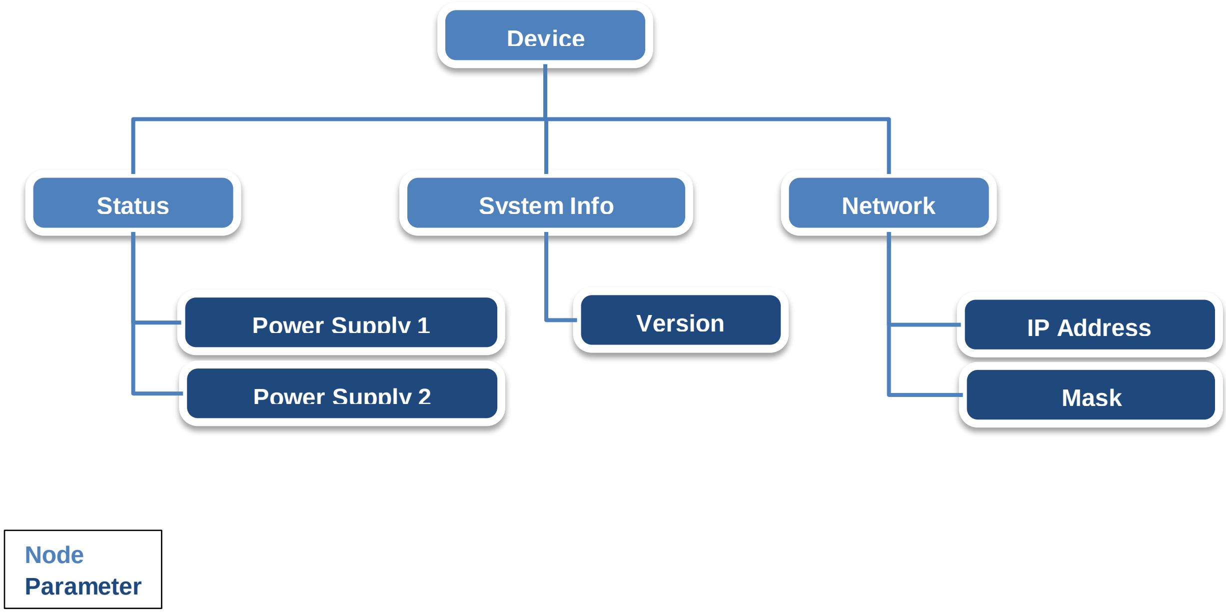
Representing a device

A common device is represented by using the Node and the Parameter type. Each node may have any number of child nodes and parameters, while a parameter may not have child nodes or parameters – although the specification technically allows it since the Children property is of type ElementCollection. But this property should only be used by a consumer to attach a command to the parameter, like Subscribe or Unsubscribe.

The sample frame provides several parameters, which are listed in the table below.

Identifier	Type	Properties
Power Supply 1	Integer (Enumeration)	▪ Value ▪ Enumeration ▪ Description ▪ Identifier
Power Supply 2	Integer (Enumeration)	▪ Value ▪ Enumeration ▪ Description ▪ Identifier
Software Version	String	▪ Value ▪ Description ▪ Identifier
IP Address	String	▪ IsWriteable ▪ Value ▪ Description ▪ Identifier
Network Mask	String	▪ IsWriteable ▪ Value ▪ Description ▪ Identifier

This is a simplification and only a subset of the parameters a device usually offers, but it is enough to show how this structure would look like when using Glow. The easiest approach would be to create a Node representing the device and then to append all parameters to it. This would work but the Node type is not restricted to representing physical entities, it may also be used for logical entities, like a category.



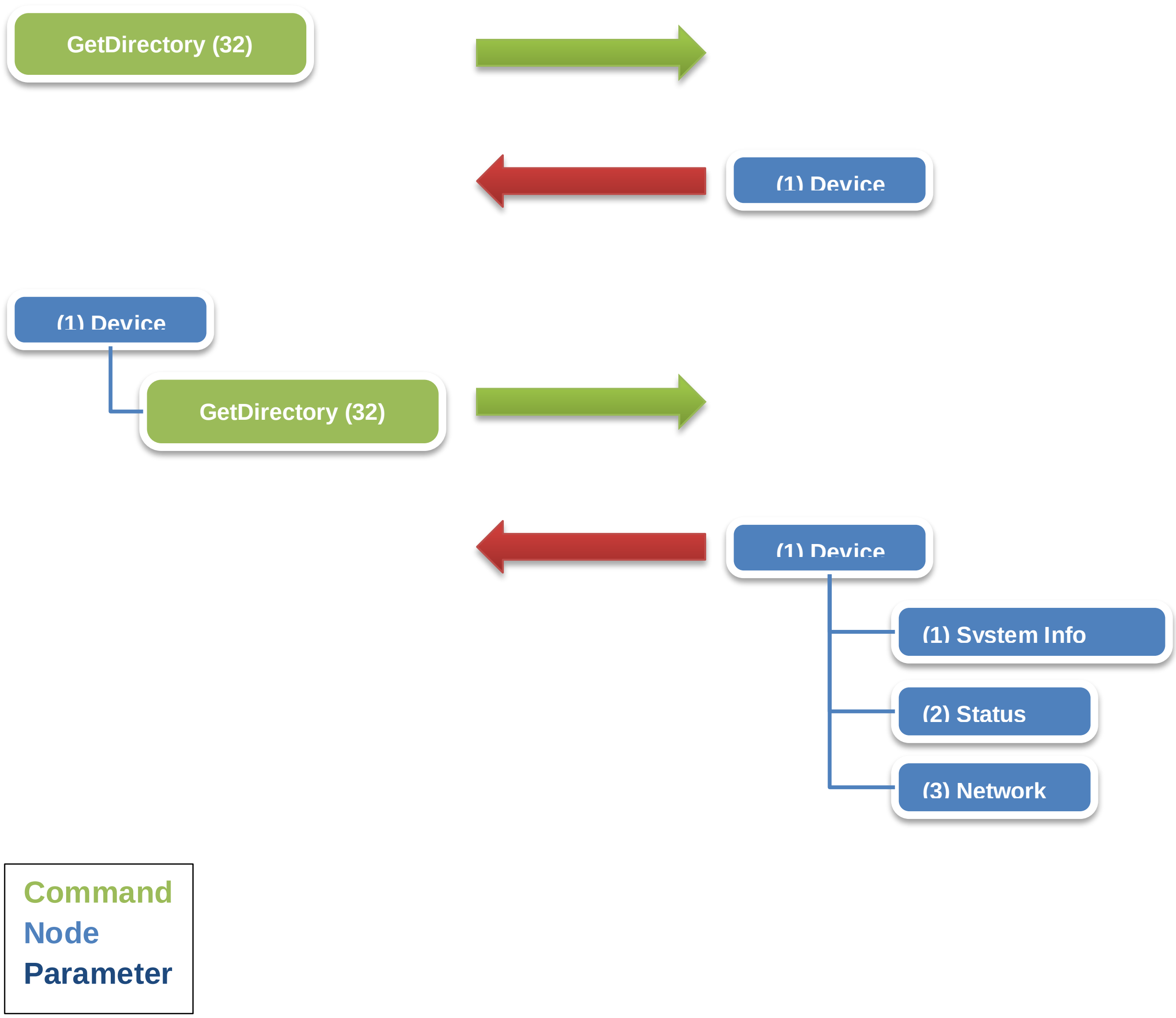
This design separates the parameters into three categories: *Status*, *System Info* and *Network*. These names are also the node identifiers. In most cases, the benefit is that the user will find the parameter it is looking for faster. A consumer may also use this information for the layout of its user interface. In bigger systems with hundreds or even thousands of parameters the use of nodes in order to categorize parameters also reduces the message sizes and reduces the CPU load of the data provider since it doesn't have to encode a large parameter set at once. Instead, it only has to transmit the parameters of the node the user or a control system is currently interested in.

Querying a data provider

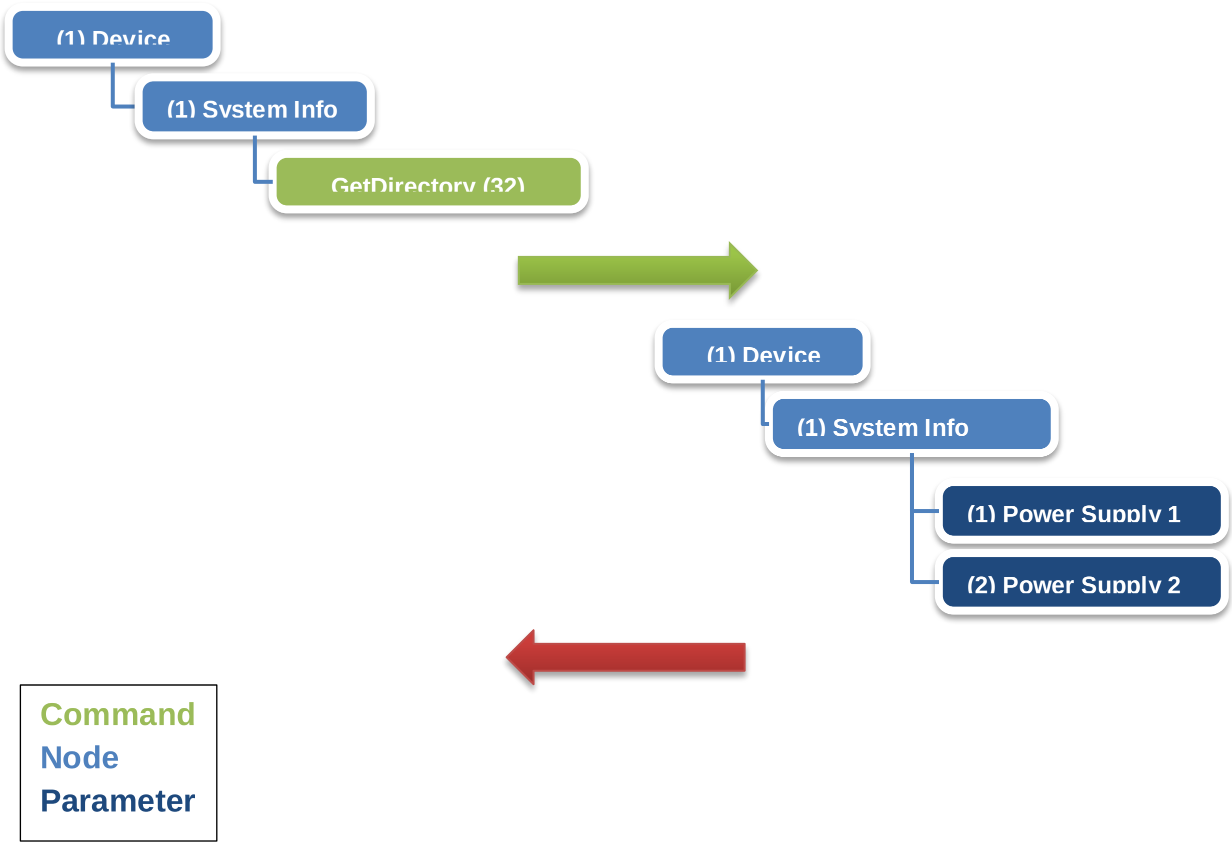
When a consumer connects to a provider it usually requests the structure of the device. This can be done by using the GetDirectory command, which has to be sent for every node the consumer is interested in.

Basic data exchange

The initial data exchange would look like this:



The consumer sends a GetDirectory request (which must be the child element of the root element collection) to the provider, which responds with another ElementCollection containing the root node “Device” with Number 1 (written within the parentheses) and its description.
To query the child elements of the device node, the consumer must create a message that contains the device node and append a GetDirectory command to the Children property of that node. The provider then responds with the device node and all of if children.
When the consumer is interested in the child elements of the “System Info”, it would send the following request:



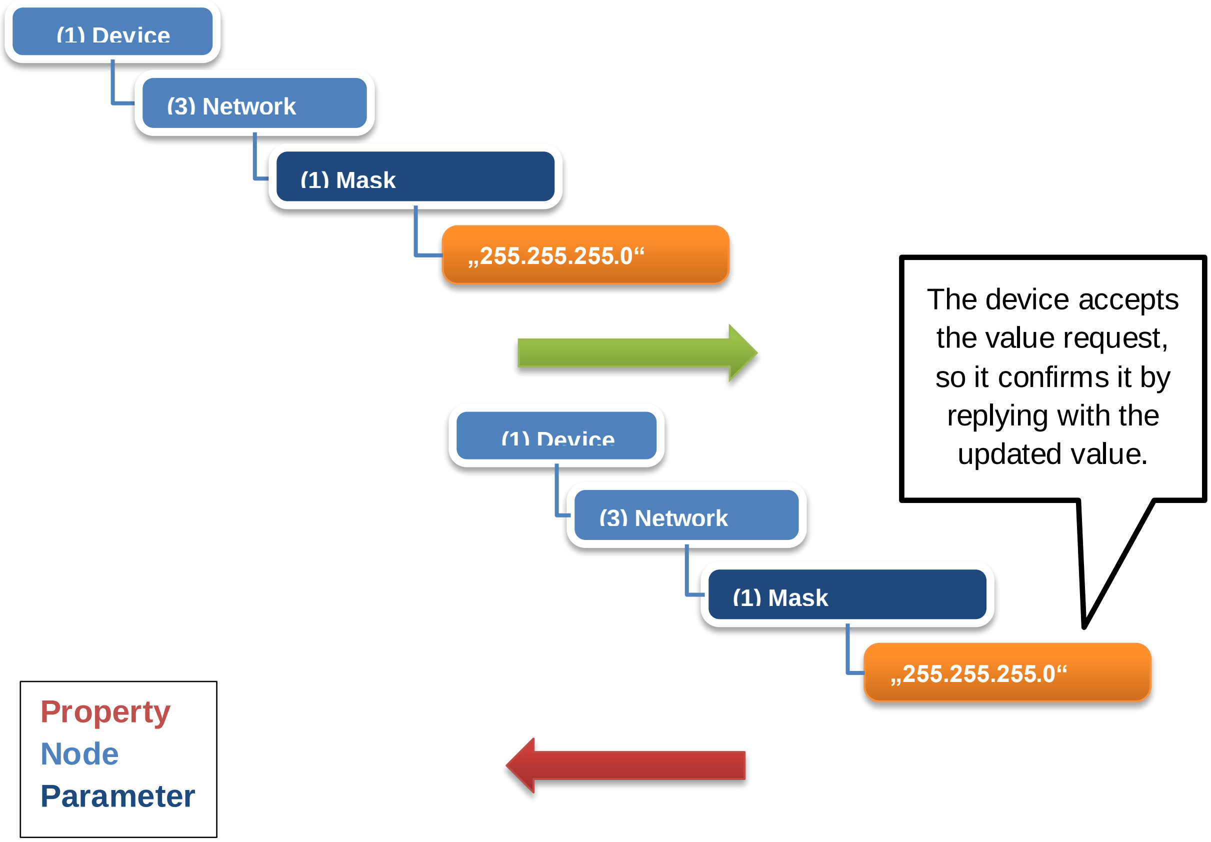
The consumer appends the "GetDirectory" command to the node it wants the get the children from, in this case "System Info". The data provider then responds with the same structure, but appends the child nodes and parameters, if there are any. When a provider reports parameters due to a "GetDirectory" request, it must include all relevant properties. That way a consumer can query the complete structure of any data provider.

Changing a parameter value

When a consumer has received the structure of a provider, a user or a control-system now has the possibility to modify the writeable values of the parameters available. There is no additional command required in order to change a parameter. Basically, the consumer only has to send the value that should be changed to the provider. The small sample structure used before has two writeable parameters: IP Address and Network Mask.

> Please note that the consumer could lose its connection to the provider when it changes the IP Address and uses TCP/IP or UDP as transport layer. This example is for demonstration purposes.

The consumer must transmit the structure containing the parameter it wants to change, and finally the value to set the parameter to. When a provider receives a value change request, it must evaluate the new value. If it is valid the provider must apply the requested change and respond with the new value. Otherwise, if the value received is invalid for some reason, the provider should discard the request and respond with the current value, even if it doesn't change. The following sample shows how a value change request looks like.



The EmBER library

The implementation of the EmBER library is written in standard C++ 03 and has been tested on several different compilers (MSVC++, g++) and platforms (Windows, Mac OS X). This section describes the different layers of the library and how to use it.
The library has been designed to work on most embedded systems as well.

Compiling the library

The library can be built with any C++ 03 standard compliant compiler. There are two variants, it may either be compiled or used as library, or it can be used header-only, which means that an application only includes the necessary headers.
To use the library as header-only, define the "LIBEMBER_HEADER_ONLY" macro before including the header file(s):

```
#define LIBEMBER_HEADER_ONLY
#include "ember/Ember.hpp"

// …
```

A premake file is provided with the library which can be used to compile the source code when not using the header-only variant. For more information, please visit:
http://industriousone.com/premake

Including the library

Each layer/namespace provides a convenience header, which should be used instead of including the files separately. If all layers are being used, the file "Ember.hpp" can be included. Otherwise, the files listed in the table below can be used:

Namespace	File
ber	"ber/Ber.hpp"
dom	"dom/Dom.hpp"
glow	"glow/Glow.hpp"

Library structure

The library has been divided into four different layers, as listed in the table below.

Util
The util namespace contains several helper classes and macros, especially the TypeErasedIterator and the OctetStream.

BER

The BER layer is the low level layer which provides the basic encoding and decoding routines for all data types supported, including the length type and the tag. The implementation uses traits classes in order to provide the encoding and decoding algorithms. That way it is easy to extend the library by additional data types.

Dom

The Dom layer provides the classes that are used to build an encodable data structure.

Glow

The Glow layer is based on the Dom layer and provides classes for each object type defined by the Glow specification, like GlowNode, GlowParameter, GlowCommand and GlowElementCollection.

The util namespace

The util namespace contains some macros and helper classes which are used all over the library, like the TypeErasedIterator and the StreamBuffer. The iterator class is used by the container classes implemented in the dom and glow namespace and allows to traverse the children of a container. The StreamBuffer is used by the encoding and decoding functions. The StreamBuffer uses a singly linked list where each node allocates a fixed size chunk.

The BER namespace

As mentioned in the introduction, this namespace contains the encoding and decoding functions. It also provides classes for the Tag, the Length and a generic Value.
To encode a value, for example an integer, its universal tag and the encoded length must be known. The universal tag contains the type information, for an integer this would be “Universal - 2”. All universal types are enumerated in the Type class (see *ember/ber/Type.h*).
The following example demonstrates how a value is being encoded.

```
util::OctetStream stream;
ber::Tag const appTag = ber::make_tag(ber::Class::Application, 1);

int const intValue = 1333;
std::size_t const frameLength = ber::encodedFrameLength(intValue);

ber::encode(stream, appTag);
ber::encode(stream, ber::make_length(frameLength));
ber::encodeFrame(stream, intValue);
```

The output buffer contains the following encoded bytes:

0x41 Application Tag “Application-1”

0x04 Inner Frame Length, 4 Bytes

0x02 Type Tag “Universal-2” (integer)

0x02 Data Length, 2 Bytes

0x05 0x35 Encoded value, 1333

To decode this data, the following sample can be used:

```
appTag = ber::decode<ber::Tag>(stream);
frameLength = ber::decode<ber::Length<int>>(stream);
universalTag = ber::decode<ber::Tag>(stream);
length = ber::decode<ber::Length<int>>(stream).value;
intValue = ber::decode<int>(stream, length);
```

This example is very basic and only demonstrates the basic use of the library and how the encoded result looks like.
In a real world example, it would be better to use the DecoderFactory class instead. It decodes a set of type, length and value and returns an instance of the type-erased Value class.

The Dom namespace

The Dom namespace offers a set of classes that can be used to create an encodable structure. A programmer using these classes does not have to have detailed knowledge about the encoding and decoding rules.

The namespace also contains two classes called DomReader and NodeFactory. The DomReader is used to decode a buffer which contains a complete tree. Additionally, it may be used to decode data which uses application defined types, like the ones used in the Glow specification. This is why the reader requires an instance of a NodeFactory, which is an abstract class responsible for creating concrete objects. It is always being called by the reader when it detects an application defined object.

The Glow namespace

The classes inside this namespace are built upon the dom classes. They represent the object types used by the Glow specification. The classes within this namespace allow it to easily construct a valid Glow message and they can also be used to traverse a decoded ember message.

Using the glow classes

This section briefly demonstrates the use of the Glow classes by providing some simple examples.

Creating a GetDirectory request

The first example shows how to generate a GetDirectory request at root level.

```
root = GlowRootElementCollection::create();

// The create method returns a collection with the application tag
// set to GlowTags::Root.

command = new GlowCommand(CommandType::GetDirectory);

root->insertElement(command);

// Encode and transmit
// …
```

The next example shows how to query the children of a specific node. It uses the small example from the previous chapters. This snippet queries the children of the Network node. The complete path up to the node that is being queried must be provided, but it is sufficient to add the node numbers. The last node then contains the GetDirectory command as child element.

```
root = new GlowRootElementCollection::create();
device = new GlowNode(1);
network = new GlowNode(3);

command = new GlowCommand(CommandType::GetDirectory);

root->insertElement(device);
device->children()->insertElement(network);
network->children()->insertElement(command);
```

It is also possible to query several nodes within one message. A consumer could construct a single tree with all known nodes and append a GetDirectory command to each of them.

Replying to a GetDirectory request

When a consumer receives a valid GetDirectory request it must respond with all nodes that belong to the one being queried. In our example, the network node contains two parameters, IP Address and Mask. So the response to the previous query would look like this:

```
root = GlowRootElementCollection::create();
device = new GlowNode(1);
network = new GlowNode(3);

ipAddress = new GlowParameter(1);
ipAddress->setValue(currentIpAddress);
ipAddress->setAccess(Access::ReadWrite);
```

```
ipAddress->setIdentifier("ipaddr");
ipAddress->setDescription("IP Address");

mask = new GlowParameter(2);
mask->setValue(currentMask);
mask->setAccess(Access::ReadWrite);
mask->setIdentifier("netmask");
mask->setDescription("Network Mask");

root->insertElement(device);
device->children()->insertElement(network);
network->children()->insertElement(ipAddress);
network->children()->insertElement(mask);
```

This way a message is being created which contains the current state of the parameters and their description.

Changing a parameter value

When a consumer wants to change a parameter value, it has to provide the partial tree and the new value. The following sample shows how a change network mask request looks like.

```
root = GlowRootElementCollection::create();
device = new GlowNode(1);
network = new GlowNode(3);

mask = new GlowParameter(2);
mask->setValue("255.255.252.0");

root->insertElement(device);
device->children()->insertElement(network);
network->children()->insertElement(mask);
```

This is it. When a provider detects a parameter containing a value, it must treat this value as value change request.

Reporting a parameter value change

When a provider receives a value change request or changes a value for some other reason, it must report the change to all connected clients. The response to the previous example looks like this:

```
root = GlowRootElementCollection::create();
device = new GlowNode(1);
network = new GlowNode(3);

mask = new GlowParameter(2);
mask->setValue(currentMask);

root->insertElement(device);
device->children()->insertElement(network);
```

```
network->children()->insertElement(mask);
```

When the consumer transmits a valid value, the response should always look like the request. If the value provided by the consumer is invalid, the provider answers with the current value.

Traversing a tree

Both, the consumer and the provider must traverse a tree when they receive it. In both cases, the procedure is very similar. This section demonstrates how a provider may traverse a tree. The sample covers Nodes, Commands and Parameters.

```
void traverseChildren(GlowElementCollection& collection)
{
   GlowElementCollection::iterator it = collection.begin();
   GlowElementCollection::iterator last = collection.end();

   for(; it != last; ++it)
   {
      dom::Node& node = *it;

      // Converts the node's type tag into a ber type. That makes it easier
      // to determine if this is an application defined type.
      ber::Type type = ber::Type::fromTag(node.typeTag());

      if(type.isApplicationDefined())
      {
         switch(type.value())
         {
            case GlowType::Node:
               handleNode(dynamic_cast<GlowNode&>(node));
               break;
            case GlowType::Parameter:
               handleParameter(dynamic_cast<GlowParameter&>(node));
               break;
            case GlowType::Command:
               handleCommand(dynamic_cast<GlowCommand&>(node));
               break;
         }
      }
   }
}

void handleNode(GlowNode& node)
{
   // An implementation should validate if the current node
   // really exists by checking its number.

   // Then, traverse the children - if there are any.
   GlowElementCollection* children = node.children();

   if(children != 0)
      traverse(*children);
```

```
}

void handleParameter(GlowParameter& parameter)
{
   // If this is a provider side implementation, the
   // provider usually only has to check if a value is set.
   // If so, this is a value request that must be handled:

   if(parameter.contains(GlowProperty::Value))
   {
      // Handle Set-Value request:
      handleParameterValue(parameter, parameter.value());
   }

   // On the other side, if this is a consumer implementation, it should check
   // the availability of each property and update its internal state.

   if(parameter.contains(GlowProperty::Access))
   {
      // modifyAccessparameter, parameter.access()
   }

   if(parameter.contains(GlowProperty::Minimum))
   {
      // modifyMinimum parameter, parameter.minimum()
   }

   // and so on . . .
}
```

Behaviour rules and other guidelines

This chapter lists the general rules a device must follow when it uses the EmBER protocol. Additionally, this chapter provides some samples that demonstrate how a tree should look like – or how it shouldn't look like.

General behaviour rules

Message length

To avoid packets that may be too large for a provider or consumer with lower memory capabilities, the size of an EmBER packet should be limited to 1024 bytes. The ember libraries provide methods to transmit partially encoded data. The framing protocol, which is used to transport the encoded EmBER data, provides a set of flags which are being used to indicate whether the encoded data is only a part of a message or a complete packet. More details about these flags and S101 in general can be found under S101 Messages.
The *StreamBuffer* class from the C++ library for example provides a virtual flush method, which is invoked whenever the buffer reaches its maximum size, which can be determined via a template parameter. When overriding this method, it is possible to encode transmit the current data before the buffer content is being reset.
That way, the memory usage is kept low and the data is being written to the clients while the EmBER encoding is in process.
On the side, when receiving a partial message that needs to be decoded, the library provides the *AsyncDomReader* class, which invokes the virtual rootReady method as soon as the decoding is complete. It is possible to derive from that class and override this method in order to process the decoded tree.

The identifier property

The identifier of a node must never change. Otherwise a consumer cannot refer to a specific node or parameter any more. If, for example, a device has several slots (1-10) which may contain different kinds of modules, there must be one unique identifier for each slot – module combination. So a node called "Slot_01" would not satisfy this requirement. The identifier should additionally contain some module information, like a type identifier or a version. The identifier could then look like "Slot_01_1000_2.0".

The description property

If a node or parameter provides a description property, it shall be displayed instead of the identifier. The identifier is not required to be human readable in a means that it has to make sense for a user.

The number property

The number is used to quickly identify a node or parameter. Once a consumer knows the structure of a data provider, it should always only provide the node or parameter numbers instead of the string identifier.

When to use the element number or identifier

Both the provider and the consumer must always include the element (node or parameter) number when transmitting a tree.
When a consumer queries the nodes of a provider by using the GetDirectory command, it must provide all attributes of the child elements, including the string identifier.
The consumer itself is never required to transmit the identifier. But if it receives an element which contains both, a number and an identifier, the identifier shall be used to map the node to an existing internal structure. This must usually only be done once when the consumer queries the provider for the first time after it established a connection.

Keep-Alive mechanism

A basic Keep-Alive mechanism can be implemented by using the S101 commands called "Keep-Alive Request" and "Keep-Alive Response". Both, the provider and the consumer may use the request command in order to test whether the remote connection is still active. The client sending the "Request-Keep Alive" message determines how long he waits for a response. A recommended timeout is about 5 seconds. Clients may close the connection to the remote session when the timeout interval expires. In general, a timeout request should not be send when the two endpoints are constantly communicating. The keep-alive request should only be sent when there hasn't been any kind of communication for a while, like for about 4-5 seconds. For more information, please refer to the chapter S101 Messages, which describes the S101 commands.

Number of consumers per provider

A provider should not limit the number of active client connections to one or two. Most modern control systems run in redundant mode and therefore it may be required that there are several connections active at once. This of course depends on the hardware capabilities of the provider, but it should be considered when designing the remote interface.

Notifications

Often a device allows several ways to be controlled, not only the remote interface. A second way would be a front panel which allows modifying some parameters. To avoid polling from the consumer side(s), a provider must always automatically notify all currently connected clients about any changes in the system. This includes the online/offline state of a node and all properties of a node or parameter.
When a parameter value changes, the provider must at least transmit the new value. All other parameter attributes must not necessarily be transmitted. However, they may be transmitted. The change notifications must be transmitted to all clients that are currently connected.

Parameter value range changes

Although it is possible to change the value range of a parameter it is not recommended. This usually irritates users working with a user interface, especially when the range changes every time when a second parameter has been modified.

Value change requests

A provider must always respond to a value change request. If the value is invalid, the provider shall respond with the current value.

General recommendations

Separate parameters into logical categories

To avoid very large parameter lists it is recommended to introduce logical categories, like “Audio”, “Video”, “Settings”.

Avoid indexed parameters

A router for example often provides a gain parameter for each source or target it has. It is NOT recommended to visualize this by providing a single node which contains all gain parameters with their number in their name. Instead there should be one node for each target or source which has the name of the target (“Target 1”) and then contains a parameter called “Gain”. The table below demonstrates both variants:

Don’t do this	Recommended variant

▪ Target Gains
    ▪ Gain [0]
    ▪ Gain [1]
    ▪ Gain [2]
    ▪ …
    ▪ Gain [N]
	▪ Targets
	    ▪ Target 1
	        ▪ Gain
	    ▪ Target 2
	        ▪ Gain
	    ▪ …

Interval for stream transmission

The recommended interval to transmit streams a consumer has subscribed to is between 50 and 80 milliseconds.

Offline trees

In several scenarios it is a requirement to store the tree structure of a provider on the consumer side. This might be the case, when a device is not accessible at any time, but its tree is required for some configuration tasks by a consumer (like a control system).

For this situation, it is possible to store the device's tree by querying all data that is required once and then storing it by using an Ember+ encoder or any user defined data format. The tree can then be accessed by the consumer as if the device was connected. It would even be possible to load the data into a kind of Ember+ emulator software. This would allow simulation of value control.
But it is important to note that the node numbers and parameters may have changed when reconnecting to the device at a later point of time. So it is important for a consumer to reference nodes by using their string identifiers instead of their number when storing a tree structure for offline use.

Frequently asked questions

This section contains a set of questions that came up during the development of the library and the integration into other projects.

Why are the node and parameter numbers session based?

The majority of the providers use a static numbering scheme for its nodes and parameters. This is the case when the hardware is known and doesn't change. But devices with modular equipment need to dynamically generate node numbers at runtime, for example when a card is being replaced by another card of a different type.

Message Framing

In order to transport the EmBER encoded data, an additional framing is required which allows the packets to be transmitted via any transport layer. This task is done by the S101 protocol, which uses a start and end byte to indicate transmission begin and end and a 16 Bit CRC which can be used to validate the packet.

The S101 protocol

To assure that the start or end byte of the framing doesn't appear within a message, all bytes with a value above 0xF8 are being escaped with a special character. First of all, the following table shows all special characters used in S101.

Name	Value	Description
BOF	0xFE	Begin of Frame
EOF	0xFF	End of Frame
CE	0xFD	Character escape
XOR	0x20	XOR value for Character escape
Invalid	0xF8	All bytes above or equal to this value must be escaped

Encoding a message

When encoding data, each data byte must be compared against the S101 Invalid value. If the value is below 0xF8, the byte can be appended to the encoding buffer. Otherwise, it must be escaped. This is done by adding the CE (0xFD) character into the stream followed by the original byte XOR'ed with XOR (0x20). Additionally each message must start with the BOF character and end with the 16 Bit CRC and the EOF character.

The data `0xFF, 0x00, 0xF9, 0x01` would encode to:

```
0xFE, 0xFD, 0xDF, 0x00, 0xFD, 0xD9, 0x01, 0x95, 0x83, 0xFF
```

Because 0xFF is above the Invalid character, it will be escaped which results in 0xFD, 0xDF. The same mechanism is applied to 0xF9. The algorithm to compute the CRC is described at the end of this chapter.

Decoding a message

When a decoder reads a BOF byte, it always indicates the start of a new frame. So the current data can be discarded if there is any. When the decoder receives a CE character, the current byte can be discarded and the next byte must be XOR'ed with 0x20 and then added to the receive buffer. When the decoder reads the EOF, the message is complete and the checksum can be evaluated.

CRC Computation

All data bytes are used for the CRC computation. The result is inverted (by applying the binary NOT operator) and the stored after the data bytes and before the EOF. The CRC bytes must also be escaped when they are equal to or above the S101 Invalid byte.
The initial value of the CRC is always 0xFFFF. The result of a decoded CRC must always be 0xF0B8. The CRC table can be found in the appendix. The following function must be applied to each data byte that is being encoded:

```
static const WORD table[256] = { . . . };

WORD ComputeCRC(WORD crc, BYTE byte)
{
   return (WORD)((crc >> 8) ^ table[(BYTE)(crc ^ byte)]);
}
```

S101 Messages

The content of all S101 messages starts with the slot identifier, which in this case is usually set to 0x00. The second byte contains the message type, which is set 0x0E for Ember. The third byte determines the command. The meaning of all other appended bytes depends on the command.

Message types

0x0E	EmBER *The range 0x00 to 0x10 is reserved for internal use.*

Commands

0x00	EmBER Packet
0x01	Keep Alive Request
0x02	Keep Alive Response

Messages

This section describes the format of the available messages.

EmBER Packet

The EmBER packet command is used whenever a device transmits data encoded with BER. The packet has the following format:

Slot	Message	Command	Version	Flags	DTD	App Bytes	<Payload>

Slot

The slot is used to address a kind of sub-device within a device. Right now, this byte should be set to 0x00.

Message

The message type; set to 0x0E.

Command

The command determines the content of the following message bytes. Available commands are EmBER (0x00), Keep-Alive Request (0x01), Keep-Alive Response (0x02).
When the command is EmBER (0x00), the payload contains an encoded Ember packet. When the command is Keep-Alive Request (0x01), the client (which may be provider or consumer) must response with a Keep-Alive Response (0x02).

Version
Set to 1.

The following fields are only available when the command type is EmBER (0x00). For Keep-Alive requests and responses, only Slot, Message type, Command type and Version must be transmitted.

Flags
The upper three bits of the flags are used to indicate partitioned ember packets. The following table lists the flags and their meaning:

0xC0	Single-packet message
0x80	First multi-packet message
0x40	Last multi-packet message
0x20	Empty packet
0x00	A packet within a multi-packet message

DTD
Defines the “Design Type Document” that is being used. This value must be set to one, which means that the Glow specification is used.

App Bytes
Defines the number of bytes that follow before the payload begins.
The current version uses two application bytes, which contain the version number of the Glow Dtd.

Payload
The BER-encoded data. This data may be decoded via a BER decoder or the DomReader.

Usage
The basic header of a non-partitioned single packet message would look like this:

```
0x00   Device Slot
0x0E   Message type: EmBER
0x00   Command type: EmBER Packet
0x01   Version
0xC0   Flags: Single Packet message (First packet | Last packet)
0x01   DTD Type: Glow DTD
0x02   Application Bytes
0x05   Minor Glow DTD Version
0x02   Major Glow DTD Version
[EmBER Data]
```

Glow DTD ASN.1 Notation

This section contains the ASN.1 notation of the Glow specification.

```
--
-- GlowDtd.asn1
-- L-S-B Broadcast Technologies GmbH
--
-- This file defines the Glow DTD used with the Ember encoding
--
-- Change Log:
-- X2.5:
--    - NOTE: This version introduces breaking changes!
--    - Changed Parameter.isCommand (BOOLEAN) to an enumeration named "type".
--      To determine the effective type of a parameter, follow this rule:
--      -
 If the parameter has either the "enumeration" or the "enumMap" field,
--        its type is "enum".
--      - If the parameter has the "value" field, its type corresponds to the
--        BER type of the value.
--      - If the parameter has the "type" field, its type is the value of this
--        field.
--      This is useful for parameters that do not specify a current value -
--      e.g. "trigger" parameters or parameters that have write-only access.
--    - Changed Parameter.isWriteable (BOOLEAN) to an enumeration named
--      "access".
--    - More options for Value - now also supports OCTET STRING and BOOLEAN
--    - Introduces QualifiedParameter and QualifiedNode types
--    - Introduces RootElement and RootElementCollection types:
--      At the root level, a different set of supported types is available.
--    - StreamCollection can also be used as root container.
--    - Introduces the StreamDescription type and the field "streamDescriptor"
--      in type ParameterContents.
-- 2.4:
--    - NOTE: This version introduces breaking changes!
--    - moved "children" in Parameter and Node out of
--      "contents" SET.
-- 2.3:
--    - Added size constraints for INTEGER values.
--    - Renamed EnumEntry to StringIntegerPair
--    - Renamed EnumCollection to StringIntegerCollection
-- 2.2:
--    - Added new field "enumMap" to Parameter and types to describe
--      enum entries: EnumEntry and EnumCollection
-- 2.1:
--    - NOTE: This version introduces breaking changes!
--    - Replaced all APPLICATION tags for fields with CONTEXT-SPECIFIC tags
--      APPLICATION tags are only used for custom types now.
-- 2.0:
--    Initial Release
--
```

```
GlowingEmber DEFINITIONS EXPLICIT TAGS ::= BEGIN


-- ======================================================
--
-- Primitive Types
--
-- ======================================================

EmberString ::= UTF8String
Integer32 ::= INTEGER (-2147483648 .. 2147483647)
Integer64 ::= INTEGER (-9223372036854775808 .. 9223372036854775807)


-- this is the base oid for all RELATIVE-OID values defined in this document.
-- when using the RELATIVE-
OID type, defining a base oid is required by ASN.1.
-- does not have any impact upon the DTD.
baseOid OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) internet(1) private(4) e
nterprises(1) lsb(37411) lsb-
mgmt(2) ember(1) emberGlow(1) glowVolatile(100) }


-- ======================================================
--
-- Parameter
--
-- ======================================================

Parameter ::=
    [APPLICATION 1] IMPLICIT
        SEQUENCE {
            number   [0] Integer32,
            contents [1] ParameterContents OPTIONAL,
            children [2] ElementCollection OPTIONAL
        }


QualifiedParameter ::=
    [APPLICATION 9] IMPLICIT
        SEQUENCE {
            path     [0] RELATIVE-OID,
            contents [1] ParameterContents OPTIONAL,
            children [2] ElementCollection OPTIONAL
        }


ParameterContents ::=
    SET {
        identifier      [ 0] EmberString            OPTIONAL,
        description     [ 1] EmberString            OPTIONAL,
        value           [ 2] Value                  OPTIONAL,
        minimum         [ 3] MinMax                 OPTIONAL,
        maximum         [ 4] MinMax                 OPTIONAL,
        access          [ 5] ParameterAccess        OPTIONAL,
        format          [ 6] EmberString            OPTIONAL,
        enumeration     [ 7] EmberString            OPTIONAL,
```

```
        factor           [ 8] Integer32                OPTIONAL,
        isOnline         [ 9] BOOLEAN                  OPTIONAL,
        formula          [10] EmberString              OPTIONAL,
        step             [11] Integer32                OPTIONAL,
        default          [12] Value                    OPTIONAL,
        type             [13] ParameterType            OPTIONAL,
        streamIdentifier [14] Integer32                OPTIONAL,
        enumMap          [15] StringIntegerCollection OPTIONAL,
        streamDescriptor [16] StreamDescription        OPTIONAL
    }


Value ::=
    CHOICE {
        integer Integer64,
        real    REAL,
        string  EmberString,
        boolean BOOLEAN,
        octets  OCTET STRING
    }


MinMax ::=
    CHOICE {
        integer Integer64,
        real    REAL
    }


ParameterType ::=
    INTEGER {
        integer (1),
        real    (2),
        string  (3),
        boolean (4),
        trigger (5),
        enum    (6),
        octets  (7)
    }


ParameterAccess ::=
    INTEGER {
        none      (0),
        read      (1),
        write     (2),
        readWrite (3)
    }


StringIntegerPair ::=
    [APPLICATION 7] IMPLICIT
        SEQUENCE {
            entryString  [0] EmberString,
            entryInteger [1] Integer32
        }
```

```
StringIntegerCollection ::=
    [APPLICATION 8] IMPLICIT
        SEQUENCE OF [0] StringIntegerPair


StreamDescription ::=
    [APPLICATION 12] IMPLICIT
        SEQUENCE {
            format [0] StreamFormat,
            offset [1] Integer32  --
 byte offset of the value in the streamed blob.
        }


-- type:       0=uint,  1=int,   2=float
-- size:       0=1byte, 1=2byte, 2=4byte, 3=8byte
-- endianness: 0=big,   1=little
StreamFormat ::=
    INTEGER {
        unsignedInt8                  ( 0), -- 00000 00 0
        unsignedInt16BigEndian        ( 2), -- 00000 01 0
        unsignedInt16LittleEndian     ( 3), -- 00000 01 1
        unsignedInt32BigEndian        ( 4), -- 00000 10 0
        unsignedInt32LittleEndian     ( 5), -- 00000 10 1
        unsignedInt64BigEndian        ( 6), -- 00000 11 0
        unsignedInt64LittleEndian     ( 7), -- 00000 11 1
        signedInt8                    ( 8), -- 00001 00 0
        signedInt16BigEndian          (10), -- 00001 01 0
        signedInt16LittleEndian       (11), -- 00001 01 1
        signedInt32BigEndian          (12), -- 00001 10 0
        signedInt32LittleEndian       (13), -- 00001 10 1
        signedInt64BigEndian          (14), -- 00001 11 0
        signedInt64LittleEndian       (15), -- 00001 11 1
        ieeeFloat32BigEndian          (20), -- 00010 10 0
        ieeeFloat32LittleEndian       (21), -- 00010 10 1
        ieeeFloat64BigEndian          (22), -- 00010 11 0
        ieeeFloat64LittleEndian       (23)  -- 00010 11 1
    }


-- ======================================================
--
-- Command
--
-- ======================================================

Command ::=
    [APPLICATION 2] IMPLICIT
        SEQUENCE {
            number    [0] CommandType,
            options   CHOICE {
                dirFieldMask [1] FieldFlags --
 only valid if number is getDirectory(32)
            } OPTIONAL
        }
```

```
CommandType ::=
    INTEGER {
        subscribe    (30),
        unsubscribe  (31),
        getDirectory (32)
    }


-- bit mask
FieldFlags ::=
    INTEGER {
        all         (-1),
        identifier  ( 1),
        description ( 2),
        value       ( 4)
    }


-- =====================================================
--
-- Node
--
-- =====================================================

Node ::=
    [APPLICATION 3] IMPLICIT
        SEQUENCE {
            number   [0] Integer32,
            contents [1] NodeContents      OPTIONAL,
            children [2] ElementCollection OPTIONAL
        }


QualifiedNode ::=
    [APPLICATION 10] IMPLICIT
        SEQUENCE {
            path     [0] RELATIVE-OID,
            contents [1] NodeContents      OPTIONAL,
            children [2] ElementCollection OPTIONAL
        }


NodeContents ::=
    SET {
        identifier  [0] EmberString       OPTIONAL,
        description [1] EmberString       OPTIONAL,
        isRoot      [2] BOOLEAN           OPTIONAL
    }


-- =====================================================
--
-- ElementCollection
--
-- =====================================================
```

```
ElementCollection ::=
    [APPLICATION 4] IMPLICIT
        SEQUENCE OF [0] Element


Element ::=
    CHOICE {
        parameter           Parameter,
        node                Node,
        command             Command
    }


-- =========================================================
--
-- Streams
--
-- =========================================================

StreamEntry ::=
    [APPLICATION 5] IMPLICIT
        SEQUENCE {
            streamIdentifier [0] Integer32,
            streamValue      [1] Value
        }


StreamCollection ::=
    [APPLICATION 6] IMPLICIT
        SEQUENCE OF [0] StreamEntry


-- =========================================================
--
-- Root
--
-- =========================================================

Root ::=
   [APPLICATION 0]
       CHOICE {
           elements RootElementCollection,
           streams StreamCollection
       }


RootElementCollection ::=
    [APPLICATION 11] IMPLICIT
        SEQUENCE OF [0] RootElement


RootElement ::=
    CHOICE {
        element            Element,
        qualifiedParameter QualifiedParameter,
        qualifiedNode      QualifiedNode
    }
```

```
END
```

Appendix

S101 CRC Table

```
0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5,
0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0xc974,
0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0x263a,
0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72,
0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0x17b9,
0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0x9bf1,
0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x0738,
0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70,
0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7,
0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0x7cff,
0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0xe036,
0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5,
0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd,
0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0xc134,
0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0x4d7c,
0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0x3efb,
0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232,
0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0x2e7a,
0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0x93b1,
0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0x1ff9,
0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330,
0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0x0f78
```

S101 Decoder Sample

```
Int  m_nRxBuffer = 0;            // Position within the Buffer
BYTE m_aRxBuffer[1024];          // The Buffer
bool m_bDataLinkEscape = false; // The Escape Flag

void Decoder:Read(BYTE Byte)
{
   if(Byte == S101_BOF)
   {
      if(m_nRxBuffer)
         TRACE("<Out of line BOF>");
```

```
      m_nRxBuffer = 1;
      m_bDataLinkEscape = false;
      return;
   }

   if(m_nRxBuffer == 0)
      return;

   if(Byte == S101_EOF)
   {
      if(m_nRxBuffer >= 4
      && CrcCCITT16(-1, m_aRxBuffer + 1, m_nRxBuffer - 1) == 0xF0B8)
         OnMessage(m_aRxBuffer + 1, m_nRxBuffer - 3);
      else
         TRACE("<RxData Error>");

      m_nRxBuffer = 0;
      return;
   }

   if(Byte == S101_CE)
   {
      m_bDataLinkEscape = true;
      return;
   }

   if(Byte >= S101_Invalid)
      return;

   if(m_bDataLinkEscape)
   {
      m_bDataLinkEscape = false;
      Byte ^= S101_XOR;
   }

   if(m_nRxBuffer < sizeof(m_aRxBuffer))
      m_aRxBuffer[m_nRxBuffer++] = Byte;
   else
      m_nRxBuffer = 0;
}
```

S101 Encoder Sample

```
void Encoder::Write (BYTE const* pBuffer, DWORD dwLength)
{
   BYTE *pSendBuffer = new BYTE [6 + dwLength * 2];
   DWORD dwLoop;
   DWORD dwLengthNew = 0;
   pSendBuffer[dwLengthNew++] = S101_BOF;

   WORD wCRC = ~CrcCCITT16(0xFFFF, pBuffer, dwLength);

   for(dwLoop = 0; dwLoop < (int) dwLength + 2; dwLoop++)
   {
      BYTE uData;
      if(dwLoop == (int) dwLength + 0)
```

```
         uData = (BYTE) wCRC;
      else
      if(dwLoop == (int) dwLength + 1)
         uData = (BYTE) (wCRC >> 8);
      else
         uData = pBuffer[dwLoop];

      if(uData >= S101_Invalid)
      {
         pSendBuffer[dwLengthNew++] = S101_CE;
         uData ^= S101_XOR;
      }

      pSendBuffer[dwLengthNew++] = uData;
   }

   pSendBuffer[dwLengthNew++] = S101_EOF;
   Write(pSendBuffer, dwLengthNew);
   delete [] pSendBuffer;
}
```