An Object-Oriented Implementation of a Low Level Reader Protocol (LLRP) Library

by

Fivos Constantinou

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2007

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by . . . .
John R. Williams
Associate Professor, Director LESL
Thesis Supervisor

Certified by . . . .
Abel Sanchez
Research Scientist, Laboratory of Manufacturing and Productivity
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

An Object-Oriented Implementation of a Low Level Reader Protocol (LLRP) Library

by

Fivos Constantinou

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This Master of Engineering Thesis describes the design and implementation of an object-oriented Low Level Reader Protocol (LLRP) library. LLRP is a recently released protocol which standardizes the formats and methods of communication between RFID Readers and Clients and aims to become the global standard for Reader management in RFID systems. LLRP provides a standard Reader - Client interface by defining a number of Messages in binary format, which Clients and Readers can each send and receive. This implementation uses a nested object model to represent all the Messages and Parameters defined in the LLRP specification. It also provides a serialization module, which converts Message and Parameter objects to the binary format described in the LLRP specification and vice versa. The use of this object model simplifies the implementation of Client logic and makes it easier to develop rich Client applications without having to deal with the low level details of the LLRP interface.

Thesis Supervisor: John R. Williams
Title: Associate Professor, Director LESL

Thesis Supervisor: Abel Sanchez
Title: Research Scientist, Laboratory of Manufacturing and Productivity

Acknowledgments

I would like to thank Prof. John Williams and Abel Sanchez for giving me the opportunity to work with the Auto-ID Labs over the last year and a half. I have have learned a lot during this time and I was very fortunate to have their precious guidance.

I would also like to thank my fellow labmates Sergio Herrero, Indy Yu and JinHock Ong for their valuable feedback and for creating a very lively and pleasant work environment.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Radio Frequency Identification (RFID) is an emerging technology which continues to be more widely adopted in a number of applications, such as access control, identification, automated timing and product tracking. One of the most important uses of RFID technology is in the supply chain as a mechanism to enhance the manufacturing process. During manufacturing, products are tagged with RFID tags and can be uniquely identified using the Electronic Product Code (EPC) [9], which is embeded in the tag. The EPC allows a product to be identified and tracked as it moves through the supply chain and finds its way to stores and eventually to consumers.

An increasing number of RFID systems are being deployed in order to enhance the manufacturing process by providing a mechanism for recording business events and data associated with a product. As RFID systems grow and become more complex, control and performance of RFID Reader devices becomes more critical [7]. An RFID system can consist of a large number of RFID readers, continuously feeding data into the enterprise network. Current RFID Networks lack communications standards and rely on vendor proprietary interfaces to exchange information between RFID Readers and the network with which they are connected. Many RFID Readers are connected to enterprise networks over TCP and are configured to transfer data via the proprietary network interface. The lack of standards causes difficulties in the adoption of RFID technology as it limits interoperability and introduces dependencies on vendor protocols. These difficulties suggest that a standard Reader-to-Network interface is needed to reduce complexity in RFID systems and make RFID adoption easier.

Recently, EPCGlobal has released the specification for the *Low Level Reader Protocol* (LLRP) [7], which specifies a standard interface between RFID Readers and Clients. The interface allows Clients to control and fully utilize Readers, providing means to retrieve Reader capabilities, control Reader operation, and allowing for robust status reporting and error handling. While the interface standardizes the way with which Clients interact with Readers it allows for air protocol specific operation and provides a mechanism for Reader vendors to define vendor specific extensions. Adopting the LLRP interface as a standard can, therefore, improve the adoption of RFID technology without limiting the control and performance of RFID devices. It is envisioned that LLRP will soon become the standard interface between RFID Readers and Clients and play a major role in the EPCGlobal Network [5].

This Master's Thesis describes the design and implementation of an object-oriented LLRP library. LLRP standardizes the formats and methods of communication between RFID Readers and Clients by defining a number of Messages that Readers and Clients can exchange. Messages are encoded in a binary format and transported over TCP. The Message format allows for a low level implementation which can be adopted even by RFID Readers with limited amounts of memory. However, on the Client side, which controls the operation of multiple readers, memory constraints are not likely to be a problem. It is, therefore, possible to take advantage of a richer implementation, which will make it easier to develop the Client logic while abstracting away the low level details of the LLRP interface.

My work in this Master's Thesis includes the following:

- Design of an object model to represent all Messages and Parameters defined in the LLRP specification and implementation using C# and the .NET framework.

- A serialization/deserialization module for converting objects to the LLRP binary format and vice versa.

- A simple Graphical User Interface for an LLRP Client, which can connect to a single Reader and exchange default Messages.

1.2 Overview of LLRP

LLRP is an application layer, message-oriented protocol, which standardizes the formats and procedures of communication between Clients and Readers. Using the LLRP interface the Client can retrieve and change the Reader configuration, controlling in this way the Reader's operation. The functionality provided by the interface is summarized below:

- Allows Clients to retrieve Reader device Capabilities.
- Allows Clients to control Readers to inventory, read, write tags and execute other access commands.
- Allows Clients to control the Reader device operation (e.g. Power levels, spectrum utilization, etc.)
- Provides robust status reporting and error handling
- Allows future expansions for support of additional air protocols
- Allows Reader vendors to define custom vendor-specific extensions

1.2.1 Operation

The protocol data units are called Messages and they constitute the mechanism by which the Client communicates with the Reader. The LLRP specification defines a number of Messages that the Client and Reader can each send and receive respectively. Messages can include a variable number of Parameters in order to communicate the specific details of the LLRP operation that is requested by the Message.

LLRP Messages are transported over TCP so it is first necessary to establish a TCP connection between the Client and the Reader. Clients and Readers can both initiate or accept a connection. Clients can be connected to multiple Readers, while Readers can hold a connection to only a single Client. After a TCP connection is established the Reader must send a status report Message, indicating whether or not the connection attempt was successful. Once the LLRP connection is established the Client and Reader can communicate by exchanging LLRP Messages.

For the most part, LLRP operation will involve the Client sending Messages to retrieve and update the Reader's configuration. Figure 1-1 shows a typical timeline of LLRP operations. Every Client Message defined in the LLRP specification has a corresponding Reader response and consequently, Reader Messages are primarily responses to Client Messages. Additionally, Readers can send status notifications, keepalives or respond with an error Message when they receive an unsupported message type.

1.2.2 Messages, Parameters and Fields

LLRP supports existing air protocols and provides an extensible mechanism for the addition of new air protocols in the future. The specification defines a number of Messages that Clients and Readers can each send and receive, specifying in this way with which Clients control the Reader's operation. Messages can be grouped according to functionality and allow a great level of control through the use of Parameters. Parameters are used to communicate specific details of the LLRP operation requested by a Message and constitute a flexible and extensible mechanism for Reader control.

The structure of LLRP Messages is common across all protocols. Messages contain a variable number of fields or Parameters. Fields are composed of the following basic data types:

- Bit
- Bit Array
- Byte Array

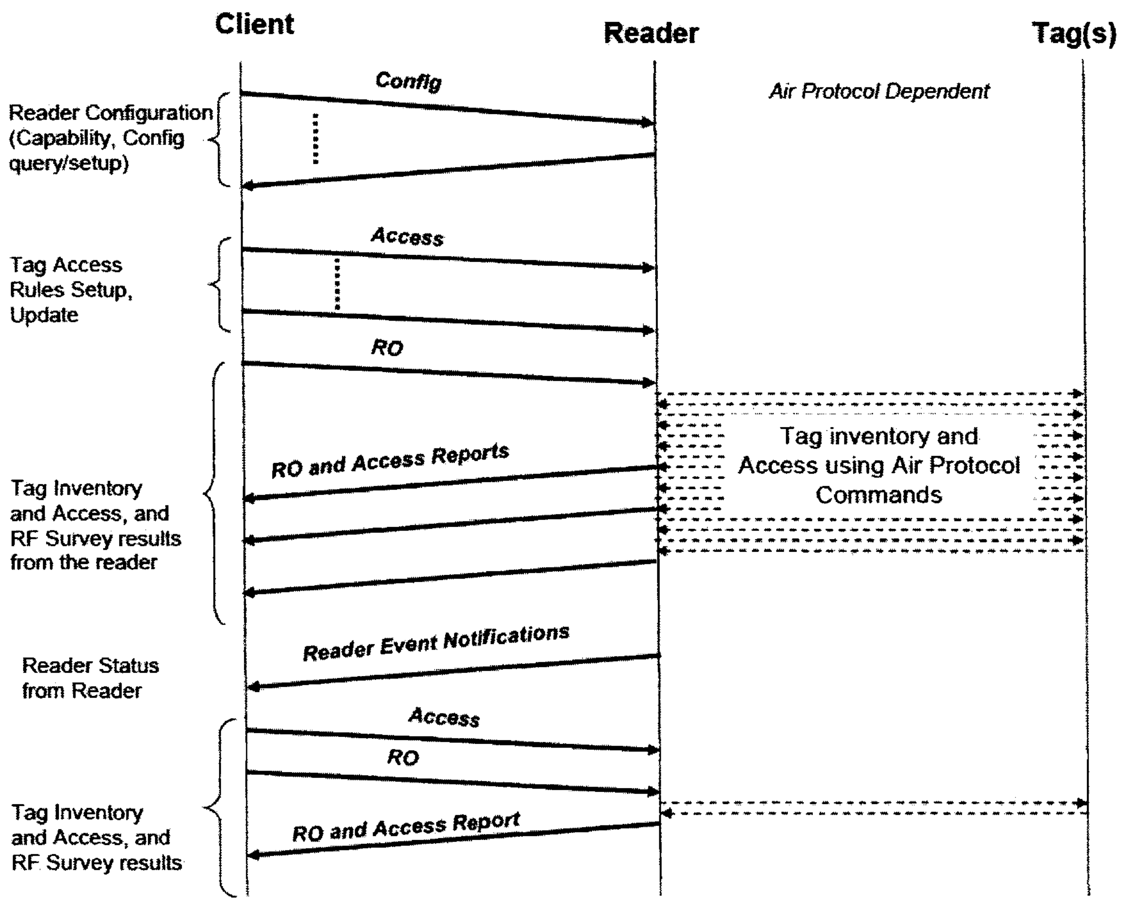

Client
Reader
Tag(s)
Config
Air Protocol Dependent
Reader Configuration (Capability, Config query/setup)
Access
Tag Access Rules Setup, Update
RO
Tag inventory and Access using Air Protocol Commands
RO and Access Reports
Tag Inventory and Access, and RF Survey results from the reader
Reader Event Notifications
Reader Status from Reader
Access
RO
Tag Inventory and Access, and RF Survey results
RO and Access Report

Figure 1-1: Typical LLRP Timeline [7]

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Rsvd			Ver			Message Type										Message Length [31:16]															
Message Length [15:0]																Message ID[31:16]															
Message ID[15:0]																															
Message Value																															

Figure 1-2: Binary Encoding of a LLRP Message

- Boolean
- Short Array
- Signed Integer
- Signed Short Integer
- Signed Long Integer
- Unsigned Integer
- Unsigned Short Integer
- Unsigned Long Integer
- UTF-8 String

Similar to Messages, Parameters have a common structure and may contain fields or sub-Parameters. It is, therefore, possible to have an arbitrarily deep nesting of Parameters.

1.2.3 Binary Encoding

LLRP Messages are encoded in a binary format to be transported between Clients and Readers over a TCP connection. The LLRP specification defines how Messages and Parameters are encoded as a stream of bytes. Figure 1-2 illustrates the binary encoding for an LLRP Message.

The details of the binary encoding for LLRP Messages are as follows:

- **Reserved Bits:** 3 bits

  The reserved bits are reserved for future extensions. Implementations of LLRP based on this specification are using the value 0.

- **Ver:** 3 bits

  The version of LLRP. Implementations of LLRP based on this specification are using the value 0x1. Other values are reserved for future use.

- **Message Type:** 10 bits

  The type of LLRP message being carried in the message.

- **Message Length:** 32 bits

  Size of the entire message in octets starting from bit offset 0 of the first word. If the Message Value field is zero-length, the Length field will be set to 10.

- **Message ID:** 32 bits

  LLRP uses a Message sequence number in each message. The Message sequence number is used to correlate a response with the original request. This sequence number is local to the LLRP channel.

- **Message Value:** variable length

  Dependent on the Message Type. Consists of a variable number of LLRP Parameters

Similar to LLRP Messages, the specification defines the binary encoding for LLRP Parameters. There are two different encodings for parameters: Type-length-value (TLV) encoded parameters and Type-value (TV) encoded parameters. TV encoding is used only for fixed length parameters, which cannot contain any sub-parameters. Figure 1-3 and figure 1-4 illustrate the binary encoding for TLV encoded and TV encoded parameters respectively.

The details of the binary encoding for TLV encoded Parameters are as follows:

- **Reserved Bits:** 6 bits

  Reserved for future extensions. All reserved bits are set to 0.

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Reserved						Parameter Type										Parameter Length															
Parameter Value																															

Figure 1-3: Binary Encoding of a TLV Encoded Parameter

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
1	Parameter Type																														
Parameter Value																															

Figure 1-4: Binary Encoding of a TV Encoded Parameter

- **Parameter Type:** 10 bits

  The type of LLRP parameter being carried in the message. The parameter number space for the TLV-parameters is 128-2047.

- **Parameter Length:** 16 bits

  The size of the entire parameter in bytes starting from bit offset 0 of the first word. If the Parameter Value field is zero-length, the Parameter Length field will be set to 4.

- **Parameter Value:** variable length

  Dependent on the Parameter Type.

The details of the binary encoding for TV encoded Parameters are as follows:

- **Parameter Type:** 8 bits

  Type of LLRP parameter being carried in the message. The parameter number space for the TV-parameters is 1-127.

- **Parameter Value:** variable length

  Dependent on the Parameter Type. Cannot contain sub-parameters

1.3 Role of LLRP in EPCGlobal Network

The EPCGlobal Network is a system designed by the Auto-ID Center at MIT together with other academic institutions and industry leaders around the world and aims to become the global standard for real-time, automatic product identification in the supply chain of any industry, anywhere in the world. The EPCGlobal Network will help increase manufacturing efficiency by providing complete visibility of information, quick product discovery and access to a product's history. It is also essential for this network to support a wide variety of business use cases that may depend on the particular industry or business sector.

The EPCGlobal Network specification [5], provides a high-level description of the various components of the network. The stack of components abstracts the different parts of the EPCGlobal network, starting from the low level tag operations all the way up to services such as EPCIS [6], Discovery and ONS [4], which are more concerned with the business context of tag reads.

Figure 1-5 shows the position of LLRP in the EPCGlobal Network.

LLRP is positioned between RFID Readers and the Filtering & Collection (F&C) applications. As shown in Figure 1-5, RFID Readers are responsible for the transmission of tag data using a Tag Protocol such as UHF Class 1 Gen 2 [3]. F&C applications filter and collect raw tag reads over time intervals and deliver the filtered and collected data to the EPCIS capturing application through the ALE [2] interface. LLRP links the Reader role with the F&C role, allowing the F&C role to control tag data collection.

1.4 Previous Work

The LLRP standard has been ratified very recently and has not yet been widely adopted. At the moment there are no implementations of LLRP publicly available but there is a lot of industry initiative to adopt LLRP as the standard Reader-Client interface.

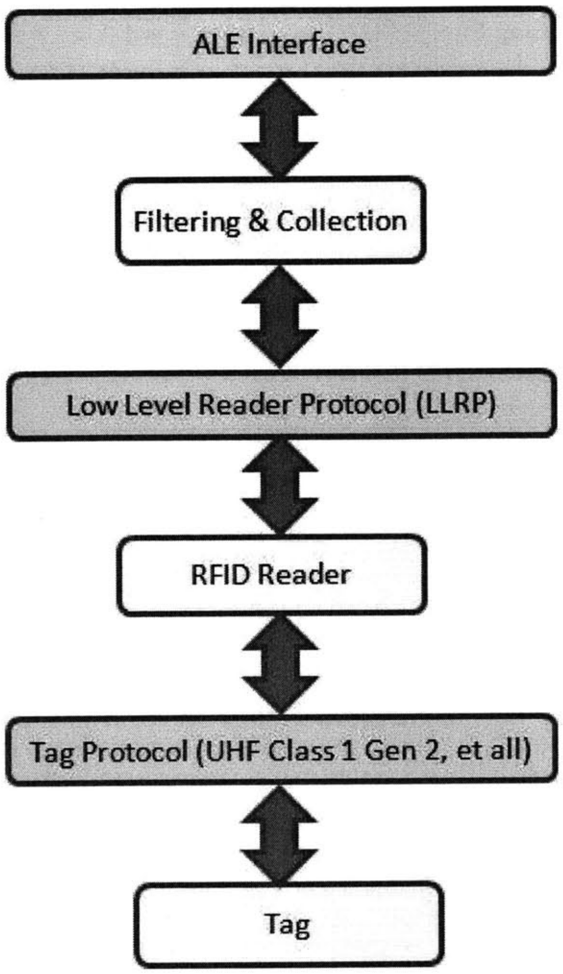

ALE Interface
Filtering & Collection
Low Level Reader Protocol (LLRP)
RFID Reader
Tag Protocol (UHF Class 1 Gen 2, et all)
Tag

Figure 1-5: Role of LLRP in EPCGlobal Network

Reader manufacturers are among the leaders in the LLRP adoption initiative. Although some Reader manufacturers have come up with their own LLRP implementations, no implementation is yet publically available. This implementation would be one of the first publically available ones and could potentially become part of a larger open source, collaborative project.

1.5 Organization of this Thesis

Chapter 2 describes the object model design and analyzes the most important design decisions. The serialization and deserialization processes are detailed in Chapter 3. Chapter 4 describes the implementation of a Graphical User Interface for an LLRP

Client. Chapter 5 discusses some of the design decisions as well as alternatives and possible future work. Finally Chapter 6 concludes this thesis.

Chapter 2

Object Model Design

Adopting LLRP as the standard RFID Reader to Client interface offers many advantages but it is, however, a challenging task. The protocol offers rich capabilities as it allows Clients to have full control over Readers, ensuring in this way that Clients can get maximum performance from devices. The protocol provides this high level of control while ensuring interoperability and allowing expansion for future air protocols or different vendor extensions. Therefore, LLRP is more complex than a protocol that is designed for more vendor specific operation.

The complexity of the protocol essentially lies in the binary nested format, which is used to represent LLRP Messages and Parameters. Messages exchanged between Readers and Clients are transported over TCP as a binary stream and can contain a variable number of Parameters. Outgoing Messages are recursively marshaled into the binary format on one end and recursively parsed on the other end. This deep marshaling/parsing is a difficult and possibly error prone task as it requires accurate manipulation of byte arrays and careful bitwise operations.

In order to abstract away the complexity of the binary nested format, this implementation of LLRP uses an object model to represent all Messages and Parameters that are defined in the LLRP specification. Every Message is represented by a class which implements the `ILlrpMessage` interface. Similarly, every Parameter is represented by a class which implements the `ILlrpParameter` interface. The details of the `ILlrpMessage` and `ILlrpParameter` classes are described in sections 2.1 and 2.2.

This implementation provides a mechanism for serializing the objects into the binary format and deserializing a binary stream to the appropriate objects. During the serialization/deserialization process an intermediate object is used to represent the binary encoding. The `MessageEncoding` class represents the binary encoding of all LLRP Messages. There are two different encodings for LLRP Parameters, thus two classes are used, `TLVParameterEncoding` and `TVParameterEncoding`. Both classes implement the `ParameterEncoding` interface. The details of `MessageEncoding` and `ParameterEncoding` classes are described in sections 2.3 and 2.4.

2.1 LLRP Messages

LLRP messages are the protocol data units by which a Client controls Reader operation. A Client can send messages to perform one of the following:

- Query Reader capabilities
- Control the Reader's air protocol inventory and RF operations
- Control the tag access operations performed by the Reader
- Query/set Reader configuration, and close the LLRP connection

Messages sent by a Client can change a Reader's state and, since state consistency is essential for the system to function properly, all Client messages must be acknowledged from the Reader. Consequently, Reader messages are primarily responses to Client messages. Additionally, Readers can send the following messages:

- Reports from Reader to Client. Reports include Reader device status, tag data, RF analysis report
- Errors. Reader responds with a generic error message when it receives an unsupported message type

An LLRP compliant Client must be capable of sending all Client messages defined in the LLRP specification and receive all Reader messages. Similarly, LLRP compliant Readers must handle Client messages properly and be able to send all Reader messages. As defined in the LLRP specification a Message contains the following:

- Version value
- Message Type
- Message ID
- A variable number of optional or mandatory Parameters

This implementation defines all LLRP messages using an object model and can, therefore, be used to implement both Client and Reader logic. Each message is implemented as a concrete class, which implements the `ILlrpMessage` interface. The structure of the Message classes, which is common for all Messages, is described below:

Each LLRP Message class contains the appropriate Parameters, as defined in the LLRP specification. Parameters are objects which implement the `ILlrpParameter` interface. A valid Message can be constructed by passing in all the mandatory and/or optional Parameters to the Message constructor. To ensure data encapsulation, Parameters are defined as private member variables and are made accessible as public properties. Properties are a built-in mechanism in C#, which allows access to the data fields through get/set methods. Message Parameters can be accessed and modified at any time after a Message is constructed.

In addition to Parameters, Message classes contain a private member variable of type `MessageEncoding`. The `MessageEncoding` class represents the binary encoding of the Message and provides methods for accessing and setting the Message header information, i.e. version value, Message type and Message ID. The details of the `MessageEncoding` class are outlined in more detail in section 2.3.

As shown in Figure 2-1, all LLRP Messages implement the `ILlrpMessage` interface, which defines a single method called `GetMessageEncoding()`. The `GetMessageEncoding()` method returns an object of type `MessageEncoding`. As

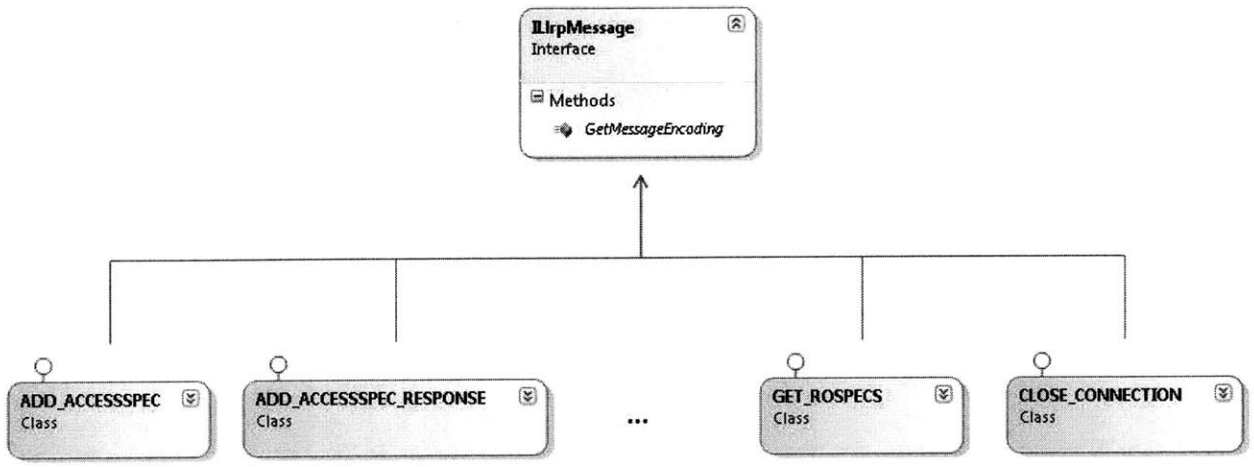


Figure 2-1: Message Classes

in the case of Parameters, data encapsulation is used for the `MessageEncoding` field. The encoding can only be retrieved by calling the `GetMessageEncoding()` method.

2.2 LLRP Parameters

LLRP Parameters are the mechanism by which Messages communicate the details of LLRP operation. As defined in the LLRP specification each Parameter contains the following:

- Parameter Type.
- Individual fields or sub-parameters.

The structure of LLRP Parameters in this implementation is similar to LLRP Messages. Each LLRP Parameter is implemented as a concrete class which implements the `ILlrpParameter` interface. Similar to Messages, Parameters contain the appropriate fields or sub-Parameters and provide the same construction and access methods.

There are two different encodings for LLRP Parameters, thus two classes to represent the binary encoding: `TLVEncoding` and `TVEncoding`. All Parameters include a field of one of the two encoding types. Both Parameter encoding classes implement

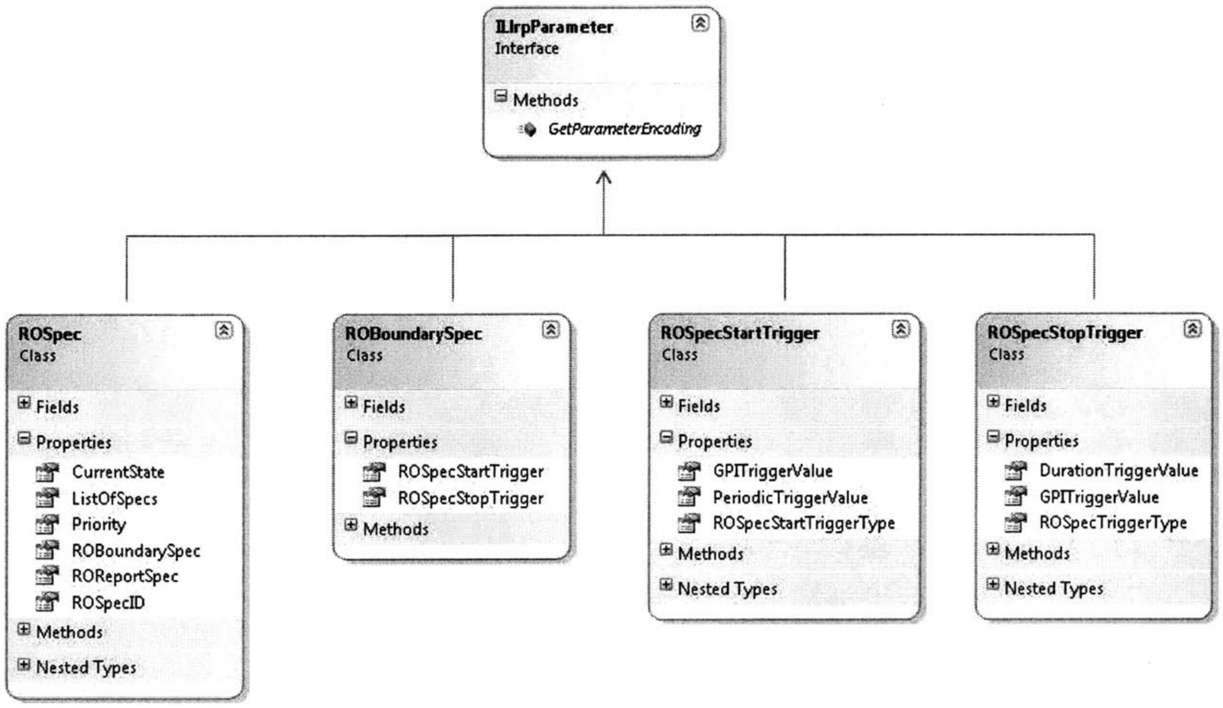


Figure 2-2: Parameter Classes

the `ParameterEncoding` interface. The details of the `TLVEncoding`, `TVEncoding` and `ParameterEncoding` classes are outlined in more detail in section 2.4.

As shown in Figure 2-2, all LLRP Parameters implement the `ILlrpParameter` interface, which defines a single method called `GetParameterEncoding()`. The `GetParameterEncoding()` method returns an object of type `ParameterEncoding`.

2.3 MessageEncoding Class

`MessageEncoding` is a generic class which can represent the binary encoding of all LLRP Messages. It is used in the serialization/deserialization process as an intermediate state between the Message Object and the binary encoded format. It is essentially an object wrapper around the binary format, providing convenient methods for accessing and setting all parts of an LLRP Message, which are outlined in section 1.2.3. Figure 2-3 shows a class diagram of the `MessageEncoding` class, illustrating the public properties for all the parts of an LLRP Message.

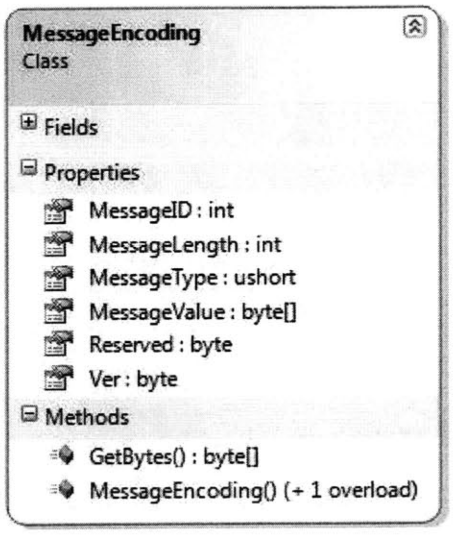


Figure 2-3: Class Diagram for the `MessageEncoding` class

The header of an LLRP Message consists of the Reserved bits, Version, Message Type, Message Length and Message ID.

In the current LLRP specification the Reserved bits are all set to 0, while Version is set to 0x1 by default. Since these values are the same for all Messages they are set as defaults during the construction of a `MessageEncoding` object.

The Message Type is an unsigned integer in the range 1-1023. Each Message class in its constructor sets its Message Type to the value specified in the LLRP specification. As shown in Figure 2-4, which shows an example of the default constructor for a Message class, the implementation provides a `MessageTypes` enumeration, which is a useful abstraction for setting the Message Type.

```
public GET_READER_CONFIG()
{
  encoding = new MessageEncoding();
  encoding.MessageType = (ushort)MessageTypes.GET_READER_CONFIG;
}
```

Figure 2-4: Example of a Message Class Constructor

The Message ID is set by the Client before sending a message to the Reader. This ID is unique for Messages sent in a single session between a Client and a Reader and is used to correlate a response with the original request.

The MessageValue property of the MessageEncoding is set by the Message class when the GetMessageEncoding() method is called. Figure 2-5 shows an example of an implementation of the GetMessageEncoding() method. Message Parameters are serialized into the binary format and merged into a single byte array, which constitutes the MessageValue. The details of the serialization process are analyzed in Chapter 3.

```
public MessageEncoding GetMessageEncoding()
{
  List<object> parameters = new List<object>();
  parameters.Add(antennaId);
  parameters.Add(requestedData);
  parameters.Add(gpiPortNum);
  parameters.Add(gpoPortNum);
  foreach (CustomParameter customParameter in customExtensionPointList)
    parameters.Add(customParameter);

  encoding.MessageValue = Serialization.GetBytes(parameters);
  return encoding;
}
```

Figure 2-5: Example of a GetMessageEncoding() method implementation

Finally, the MessageLength property is calculated dynamically by adding the lengths of the Message header and Message Value. No set method is provided for Message Length since it is a derived value.

2.4 ParameterEncoding Classes

Similar to the MessageEncoding class, which represents the binary encoding of an LLRP Message, there are two classes for representing the binary encoding for LLRP Parameters; the TLVParameter and TVParameter classes. Both classes implement the ParameterEncoding interface, which requires them to implement the GetBytes() method. All classes implementing the ILlrpParameter interface, contain a field of either TLVParameter or TVParameter type.

2.4.1 TLVParameter Class

Figure 2-6 shows all the public properties and methods of the `TLVParameterEncoding` class, which represents the binary encoding for Type-length-value (TLV) encoded parameters.

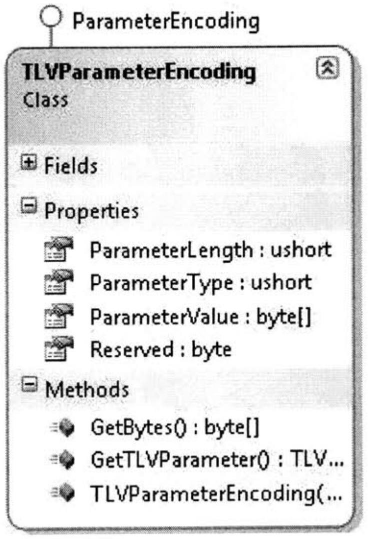


Figure 2-6: Class Diagram for the `TLVParameterEncoding` class

In the current version of the LLRP specification the Reserved bits are set to 0 by default. The Parameter class is responsible for setting the ParameterType and ParameterValue of the `TLVParameterEncoding`. Since TLV Parameters can contain a variable number of fields and sub-parameters they need to have a length property, which is dynamically set during serialization by adding the header and message value lengths.

2.4.2 TVParameter Class

Figure 2-7 shows all the public properties and methods of the `TVParameterEncoding` class, which represents the binary encoding for Type-value (TV) encoded parameters.

The Parameter class is responsible for setting the ParameterType and ParameterValue of the `TVParameterEncoding`. TV-encoded parameters cannot contain sub-parameters and thus have fixed length.

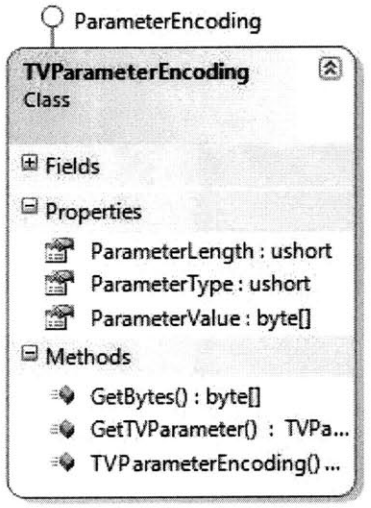


Figure 2-7: Class Diagram for the `TVParameterEncoding` class

Chapter 3

Object Serialization and Deserialization

The Client logic is implemented by making use of the object model, which is outlined in Chapter 2. In order to communicate with Readers it is necessary to have a serialization/deserialization process, which converts Message and Parameter objects to the binary format defined in the LLRP specification and vice versa. The serialization/deserialization process is outlined in Figure 3-1.

3.1 Serialization

3.1.1 Message Object to `MessageEncoding`

The serialization process from a Message Object to the binary format includes an intermediate step, namely the conversion to a `MessageEncoding` object. As described in Section 2.3, `MessageEncoding` is a generic class which can represent the binary encoding of all LLRP Messages and is used as a convenient object wrapper around the binary encoding format. The first step in the serialization process is to to get a `MessageEncoding` object by calling the Message's `GetMessageEncoding()` method.

The `GetMessageEncoding()` method returns a `MessageEncoding` object after setting its `MessageValue` property. The `MessageValue` is a byte array consisting of all

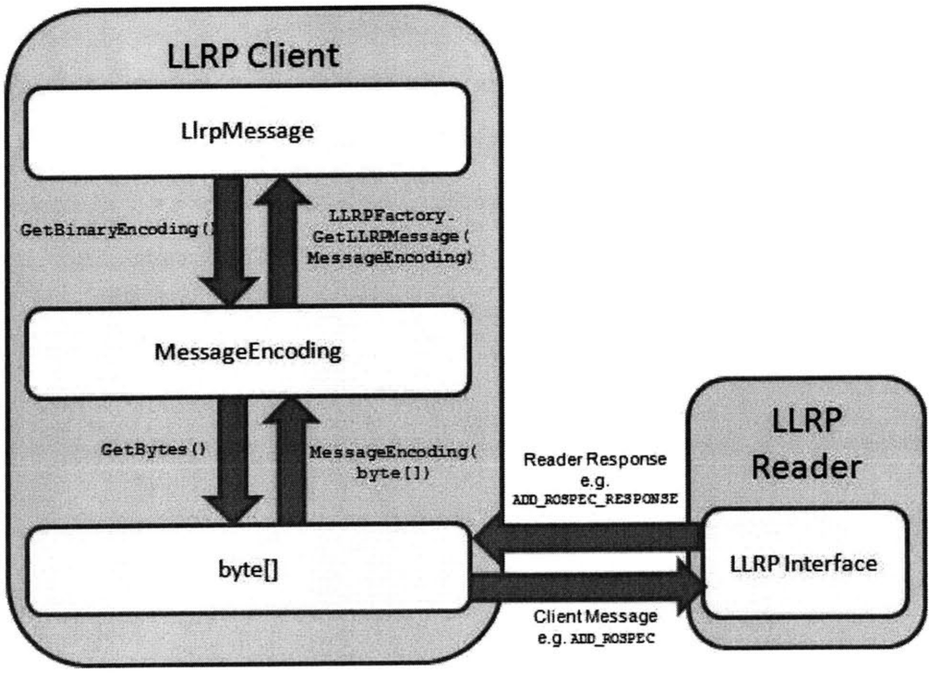
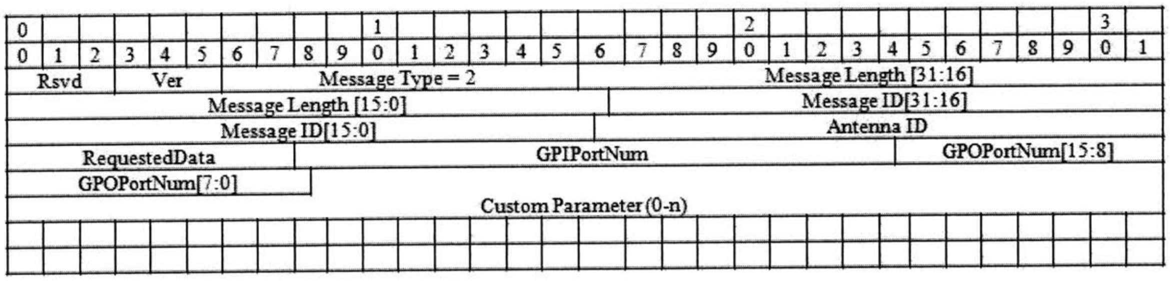


Figure 3-1: Overview of the Serialization Process

Message Parameters, each binary encoded according to the LLRP specification. Figure 3-2 shows the binary encoding of a `GET_READER_CONFIG` Message as an example.



0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1

Rsvd	Ver	Message Type = 2	Message Length [31:16]
Message Length [15:0]			Message ID[31:16]
Message ID[15:0]			Antenna ID
RequestedData	GPIPortNum		GPOPortNum[15:8]
GPOPortNum[7:0]	Custom Parameter (0-n)		

Figure 3-2: Binary encoding for a GET_READER_CONFIG Message

Using Serialization helper methods each Parameter is converted into a byte array and eventually all byte arrays are merged into a single byte array, which makes up the Message Value. Figure 3-3 shows part of the `GET_READER_CONFIG` class to illustrate how serialization takes place in the `GetMessageEncoding()` method.

As shown in Figure 3-3 in the `GetMessageEncoding` method all Parameters are inserted into a List in the order specified by the LLRP specification. The List is then

```
public class GET_READER_CONFIG : ILLRPMessage
{
  ushort antennaId;
  byte requestedData;
  ushort gpiPortNum;
  ushort gpoPortNum;
  List<CustomParameter> customExtensionPointList;
  MessageEncoding encoding;

  public GET_READER_CONFIG()
  {
  encoding = new MessageEncoding();
    encoding.MessageType = (ushort)MessageTypes.GET_READER_CONFIG;
  }

  public MessageEncoding GetMessageEncoding()
  {
    List<object> parameters = new List<object>();
    parameters.Add(antennaId);
    parameters.Add(requestedData);
    parameters.Add(gpiPortNum);
    parameters.Add(gpoPortNum);
    foreach (CustomParameter customParameter in customExtensionPointList)
      parameters.Add(customParameter);

  encoding.MessageValue = Serialization.GetBytes(parameters);
    return encoding;
  }
}
```

Figure 3-3: Message Class Example (GET_READER_CONFIG)

passed as a Parameter to the `Serialization.GetBytes()` method, which sequentially converts each Parameter into the binary format, merges the subsequent byte arrays and returns a single byte array.

The `Serialization` class contains a number of static methods for converting both primitive types and LLRP Parameters. The `GetBytes()` method is overloaded to accept any valid Parameter type and return its binary encoding.

Integer Types

Integers are encoded in network byte order with the most significant byte of the integer sent first (Big-Endian). The following integer types supported:

Signed Integer: A 32-bit signed integer, encoded using 4 bytes.

Unsigned Integer: A 32-bit unsigned integer, encoded using 4 bytes.

Signed Short Integer: A 16-bit signed integer, encoded using 2 bytes.

Unsigned Short Integer: A 16-bit unsigned integer, encoded using 2 bytes.

Signed Long Integer: A 64-bit signed integer, encoded using 8 bytes.

Unsigned Long Integer: A 64-bit unsigned integer, encoded using 8 bytes.

Strings

Strings are encoded in UTF-8 format [10]. Since strings can have variable length, each UTF-8 encoded string is preceded by a 16-bit integer which specifies the length of the string in bytes.

Booleans

A Boolean value is represented by a single bit. LLRP, however, requires the length of all Messages to be a multiple of octets, which requires the binary encoding of all Parameters to be a multiple of octets. Consequently, Boolean parameters are padded with zeros to ensure that the length is a multiple of octets.

It is possible to have a number of Boolean values grouped together in the same byte. This implementation provides a `GetBytes()` method overload, which converts a list of Boolean values to a byte array. For example, if a list with 10 Boolean values is passed, two bytes will be required. The first 10 bits of the 2-byte array will represent the Boolean values, while the remaining bits will be set to 0.

LLRP Parameters

Since Parameters can contain a variable number of fields or sub-parameters, they are recursively converted into a binary format by calling the Parameter's `GetParameterEncoding()` method. Section 3.1.2 describes the details of Parameter Object conversion to the binary format.

Lists and Arrays

In addition to basic types, fields can be defined as Lists or arrays of primitive or sub-parameter types. Lists are encoded by iterating through their elements and encoding each element in order.

3.1.2 Parameter Object to `ParameterEncoding`

Similar to Messages, which provide a `GetMessageEncoding()` method, all classes implementing the `ILlrpParameter` interface must implement the `GetParameterEncoding()` method. Since Parameters can contain sub-Parameters, which in turn can contain other sub-Parameters and so on, Parameter objects might be deeply nested. By implementing the `GetParameterEncoding()` method, Parameters provide a way to go all the way down in the object hierarchy and recursively convert Parameters to a nested binary format.

The `GetParameterEncoding()` method is similar to the `GetMessageEncoding()` method. It inserts all Parameters into a List and passes it as an argument to the `Serialization.GetBytes()` method. The returned byte array constitutes the Parameter Value of the `ParameterEncoding`.

3.1.3 `MessageEncoding` to byte[]

Once the `MessageEncoding` is obtained the Client sets the MessageID property before sending the message to the Reader. The MessageID uniquely identifies a Message in a single session between a Client and a Reader and is used to associate the Reader's response with the appropriate message. The `MessageEncoding` is subsequently converted into a byte array in order to be transmitted to the Reader over the TCP stream. The `MessageEncoding` class provides a `GetBytes()` method, shown in Figure 3-4, which converts the object to the binary format.

The `GetBytes()` method sets the header bits and merges the header and message value into a single byte array. The returned byte array is a well-formed binary encoded LLRP Message, as defined in the LLPR specification.

```
/// <summary>
/// Merges the header and message value into a single byte array
/// </summary>
/// <returns>
///    A byte array, which constitutes a complete and well-formed LLRP Message
/// </returns>
public byte[] GetBytes()

  // Set Reserved Bits
  header[0] |= (byte)(reserved);

  // Set Version Bits
  header[0] |= (byte)(ver << 3);

  // Set Message Type
  byte[] bits = Serialization.UShortToByteArray(messageType);
  header[0] |= (byte)(bits[0] << 6);
  header[1] |= bits[1];

  // Set MessageID
  Serialization.CopyIntToByteArray(header, MESSAGEID_BYTE_OFFSET, messageId);

  // Set Message Length
  Serialization.CopyIntToByteArray(header, MESSAGE_LENGTH_BYTE_OFFSET, MessageLength);

  byte[] message = new byte[MessageLength];

  // Copy header and message value arrays to a new byte array
  header.CopyTo(message, 0);
  messageValue.CopyTo(message, header.Length);

  return message;
```

Figure 3-4: GetBytes method in `MessageEncoding` class

3.1.4 `ParameterEncoding` to byte[]

Similar to `MessageEncoding`, `ParameterEncoding` classes provide a `GetBytes()` method for converting the encoding object into the binary format. Figures 3-5 and 3-6 show the `GetBytes()` methods for the two types of `ParameterEncoding`, `TLVParameterEncoding` and `TVParameterEncoding` respectively.

```
/// <summary>
/// Merges the header and parameter value into a single byte array
/// </summary>
/// <returns>
/// A byte array, which constitutes a complete and well formed LLRP Parameter
/// </returns>
public byte[] GetBytes()

  // Set Reserved Bits
  header[0] |= (byte) reserved;

  // Set Parameter Type
  byte[] bits = Serialization.GetBytes(parameterType);

  header[0] |= (byte) (bits[0] << 6);
  header[1] |= bits[1];

  // Set Parameter Length
  Serialization.InsertBytes(header, PARAMETER_LENGTH_BYTE_OFFSET, ParameterLength);

  byte[] parameter = new byte[ParameterLength];

  // Copy header and message value arrays to a new byte array
  header.CopyTo(parameter, 0);
  parameterValue.CopyTo(parameter, header.Length);

  return parameter;
```

Figure 3-5: GetBytes method in `TLVParameterEncoding` class

3.2 Deserialization

The Client receives Reader Messages in binary format over the established TCP connection and deserialized them into objects. The deserialization process is described below.

3.2.1 Byte[] to `MessageEncoding`

Reader Messages arrive over a TCP stream and have a variable byte length. As the Client reads bytes off the TCP stream it needs to determine the message length and read the exact number of bytes off the stream. Therefore, the Client first reads the Message header, which specifies the Message's length in bytes, and proceeds to read

```
/// <summary>
/// Merges the header and parameter value into a single byte array
/// </summary>
/// <returns>
/// A byte array, which constitutes a complete and well formed LLRP Parameter
/// </returns>
public byte[] GetBytes()

  // Set the most significant bit of the header to 1.
  header[0] |= (byte)(0x1);

  // Set Parameter Type
  header[0] |= (byte)(parameterType << 1);

  int parameterLength = header.Length + parameterValue.Length;
  byte[] parameter = new byte[parameterLength];

  // Copy header and message value arrays to a new byte array
  header.CopyTo(parameter, 0);
  parameterValue.CopyTo(parameter, header.Length);

  return parameter;
```

Figure 3-6: GetBytes method in `TVParameterEncoding` class

the Message value, which consists of the bytes remaining after the header.

The Client constructs a `MessageEncoding` object by passing the header and message value byte arrays to the `MessageEncoding` constructor. The constructor goes through the header array and sets the Reserved bits, version, Message Type and Message ID of the `MessageEncoding` object. The message value argument is used to set the `MessageValue` property of the the `MessageEncoding` object.

The `MessageEncoding` object can then be used to construct the appropriate `ILlrpMessage`.

3.2.2 Message Object Construction from `MessageEncoding`

This implementation makes use of the Factory Design Pattern [8] to get around the problem of creating objects without knowing the exact class of the object that is created. The `LlrpFactory` class, which implements the Factory Design Pattern, provides a GetILlrpMessage() method which is called with a MessageEncoding as a parameter and based on the Message Type calls the corresponding Message constructor, as shown in Figure 3-7.

```
public static ILlrpMessage GetILlrpMessage(MessageEncoding encoding)
{
  switch (encoding.MessageType)
  {
    case (ushort)MessageTypes.GET_READER_CAPABILITIES:
      return new GET_READER_CAPABILITIES(encoding);
    case (ushort)MessageTypes.GET_READER_CAPABILITIES_RESPONSE:
      return new GET_READER_CAPABILITIES_RESPONSE(encoding);
    case (ushort)MessageTypes.ADD_ROSPEC:
      return new ADD_ROSPEC(encoding);
    case (ushort)MessageTypes.ADD_ROSPEC_RESPONSE:
      return new ADD_ROSPEC_RESPONSE(encoding);

    ...

    default:
      throw new ArgumentException("Invalid Message Encoding");
  }
}
```

Figure 3-7: GetILlrpMessage Method

All Message classes provide a constructor which takes a MessageEncoding as an argument. This constructor uses Serialization methods to parse and construct the Message Parameters from the MessageValue of the MessageEncoding. In order to parse Message Parameters it is necessary to explicitly know the expected Parameters and the order in which they appear in the binary encoding. Nested Parameters must be recursively parsed and optional Parameters must be handled properly.

As shown in Figure 3-8, Parameters are sequentially parsed from the MessageValue byte array and deserialized using the Serialization methods.

The Serialization class provides static methods for deserializing Parameters of all supported types. All methods take the MessageValue array and the current position in the array as arguments. An integer variable, called currentIndex, is used

```
/// <summary>
/// Constructs a GET_READER_CONFIG message from its encoding
/// </summary>
/// <param name="encoding">The encoding.</param>
public GET_READER_CONFIG(MessageEncoding encoding)
  : this()
{
  byte[] messageValue = encoding.MessageValue;
  int currentIndex = 0;

  antennaId = Serialization.ToUShort(messageValue, ref currentIndex);
  requestedData = messageValue[currentIndex++];
  gpiPortNum = Serialization.ToUShort(messageValue, ref currentIndex);
  gpoPortNum = Serialization.ToUShort(messageValue, ref currentIndex);
  customExtensionPointList =
   Serialization.ToLLRPParameterList<CustomParameter>(
     messageValue, ref currentIndex);
}
```

Figure 3-8: Constructor for GET_READER_CONFIG with `MessageEncoding` argument

to keep track of the position in the `MessageValue` array. The `currentIndex` is passed by reference and is modified by the Serialization methods so that it points to the next Parameter.

Integer Types

Parameters of integer type are converted by combining the appropriate number of bytes. For example, to deserialize a short integer we read 2 bytes from the `MessageValue` array, starting at the `currentIndex`, and combine the upper and lower bits to get the integer value. The `currentIndex` is incremented by 2 so that it points to the next Parameter in the array.

Strings

Since strings can have variable length their encoding is always preceded by its length, which is denoted by a 16-bit integer. To decode a string the value of the length is first deserialized and is used to read the exact number of bytes off the `MessageValue` array.

The .NET framework provides support for deserializing a UTF-8 encoded string.

LLRP Parameters

Complex Parameters are constructed using the `ToILlrpParameter()` method. Since Parameters can be nested they are recursively deserialized from the nested binary format. The details are analyzed in sections 3.2.3 and 3.2.4.

Lists and Arrays

Lists and arrays are deserialized sequentially, essentially reversing their serialization process. Lists and arrays of basic types are always preceded by the number of elements in the list, which is required in order to read the correct number of bytes off the `MessageValue` array.

3.2.3 Byte[] to `ParameterEncoding`

LLRP Parameters are constructed by sequentially processing the `MessageValue` byte array and recursively parsing the binary nested format. Before constructing a Parameter object `ParameterEncoding` object is first constructed. Since there are two types of `ParameterEncodings` it is first necessary to determine which of the two encoding types is being deserialized. This can be done by checking the first bit of the header. All TLV-encoded Parameters have a 0 in the first bit of the header while TV-encoded Parameters have the first bit set to 1.

The `TLVParameterEncoding` and `TVParameterEncoding` classes provide static methods, `GetTLVParameter` and `GetTVParameter` respectively, each returning an encoding object.

The `GetTLVParameter` method first reads the header, which specifies the length of the Parameter in bytes. Once the Parameter length is calculated a `TLVParameterEncoding` object can be constructed by passing the Parameter header and Parameter Value as constructor arguments.

The `GetTVParameter` method reads the header, which only includes the Parameter Type since TV-encoded Parameters have fixed length. Parameter classes for TV-encoded Parameters include a static property which indicates the fixed Parameter length. In order to construct a `TVParameterEncoding` object it is necessary to determine the Parameter type, get the Parameter length and read the exact number of bytes off the array.

3.2.4 Parameter Object Construction from `ParameterEncoding`

Parameter object construction makes use of the Factory Design Pattern, as in the case of Messages. The `LlrpFactory` class provides a `GetILlrpParameter` method which can accept either a `TLVParameterEncoding` or a `TVParameterEncoding` as an argument and construct the corresponding Parameter object. The `GetILlrpParameter` method is called internally by serialization methods during the construction of `LlrpMessages` from their `MessageEncoding`.

All Parameter classes provide a constructor which takes either a `TLVParameterEncoding` or a `TVParameterEncoding` as an argument. The constructor uses Serialization methods to parse and construct the sub-Parameters from the Parameter Value.

Parameter deserialization makes use of the built-in C# generics. `ToILlrpParameter<>()` is a generic method which requires the type of the expected Parameter to be specified. For example, if an `ROSpec` Parameter is expected the `ToILlrpParameter<ROSpec>()` method is called, which will return an `ROSpec` object. Using generics it's possible to have the return type of a method dynamically specified and allow a single method to handle all Parameter types.

The `ToILlrpParameter<>()` method constructs a Parameter and checks if its type matches the expected type. This is a necessary step to handle Parameters that are either optional or can appear multiple times in a Message.

In the case of optional Parameters, the method attempts to construct the expected Parameter. If a different Parameter is constructed or if we reach the end of the input array, a null value is returned and the array index is not incremented.

In the case when Parameters can appear multiple times, a List is constructed to hold Parameters of the expected type. The `ToILlrpParameter<>()` method is called in a loop until we get a Parameter of a different type or we reach the end of the array. If no Parameters of the expected type are found then an empty list is returned.

Chapter 4

A Graphical LLRP Client

As a proof of concept I have used my object oriented LLRP library to implement a prototype Graphical User Interface (GUI) for a simple LLRP Client. The Client has limited functionality but the goal is to show that the object model simplifies the development of LLRP Client applications.

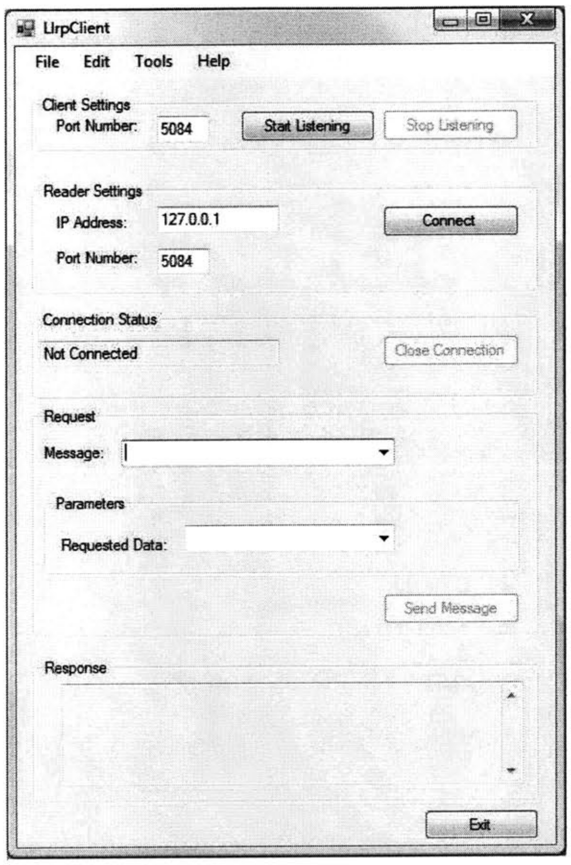
4.1 TCP Connections

The LLRP specification requires Clients and Readers to be able to both initiate or accept connections. At any given time a Client or Reader can be configured to only either initiate or accept a connection.

As shown in Figure 4-1, the Client implementation provides functionality for both initiating and accepting a connection. The Client can start listening for a connection on a port that is specified by the user. The default port defined in the LLRP specification is 5084.

The GUI also allows a user to specify the IP address and port number of a Reader and attempt to initiate a connection. In order to establish a connection the specified Reader must be listening for incoming connections.

Figures 4-2 and 4-3 show the sequence of events in the case of Reader initiated and Client initiated connections respectively. In both cases, after a TCP connection is established the Reader must reply with a status report Message. The report


LlrpClient

File Edit Tools Help

Client Settings
Port Number: 5084

Reader Settings
IP Address: 127.0.0.1
Port Number: 5084

Connection Status
Not Connected

Request
Message:

Parameters
Requested Data:

Response

Figure 4-1: LLRP Client GUI

must include a *ConnectionAttemptEvent* Parameter and should indicate connection success, if the TCP connection was established successfully.

The Client implementation uses asynchronous operations to communicate with Readers. When the Client is set to listen for incoming connections it starts an asynchronous listen operation, using the build-in asynchronous operations of the TcpListener class, which is provided by the .NET framework. The asynchronous listen operation specifies a Callback method, which is called upon a connection attempt.

The `OnConnect()` callback method is called once a TCP connection between the Client and a Reader is established. The Client then starts an asynchronous Read operation, waiting for a ReaderEventNotification Message from the Reader. If this operation times out, an incorrect Message is received, or the status report indicates an unsuccessful connection, then the TCP Connection is terminated.

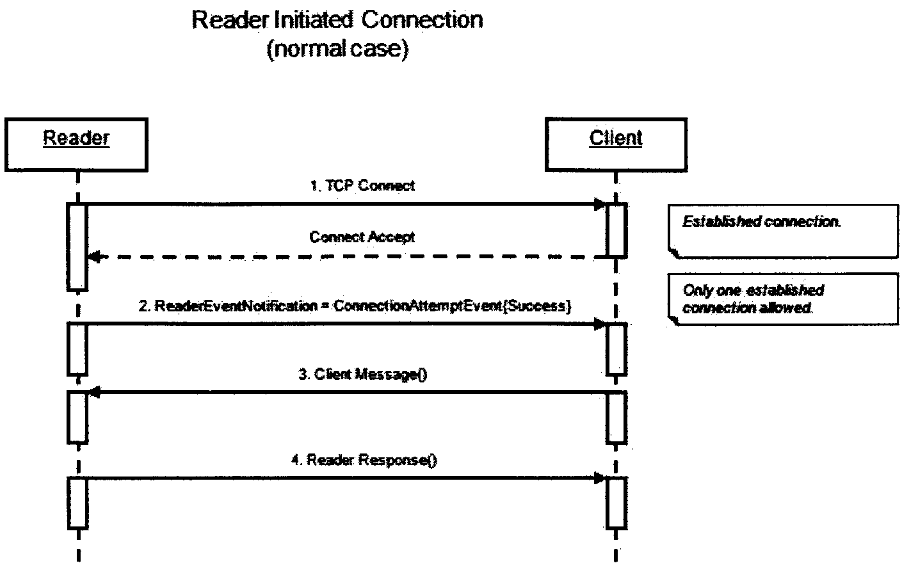

Reader Initiated Connection
(normal case)
Reader
Client
1. TCP Connect
Connect Accept
2. ReaderEventNotification = ConnectionAttemptEvent{Success}
3. Client Message()
4. Reader Response()
Established connection.
Only one established connection allowed.

Figure 4-2: Reader Initiated Connection

An LLRP Client would normally connect to and control the operation of multiple Readers. This implementation, however, currently only supports a single Reader. The GUI allows the user to initiate a connection to only a single Reader. Additionally, if the Client is in listening mode, once the Client accepts a connection it stops listening and ignores any other connection attempts.

4.2 Message Handling

As shown on Figure 4-4 the GUI allows a user to select the Message to send from a drop down menu which includes all Client Messages defined in the LLRP specification.

Depending on the selected Message, the Parameters grouping is adjusted to display the appropriate Parameters associated with the select Message. Figure 4-5 shows the Parameter configuration for a `GET_READER_CAPABILITIES` Message. The `GET_READER_CAPABILITIES` Message specifies a *Requested Data* Parameter, which can be set using a drop-down menu.

Once all Parameters are set the Message can be send to the Reader by clicking the *Send Message* button, as shown in Figure 4-6

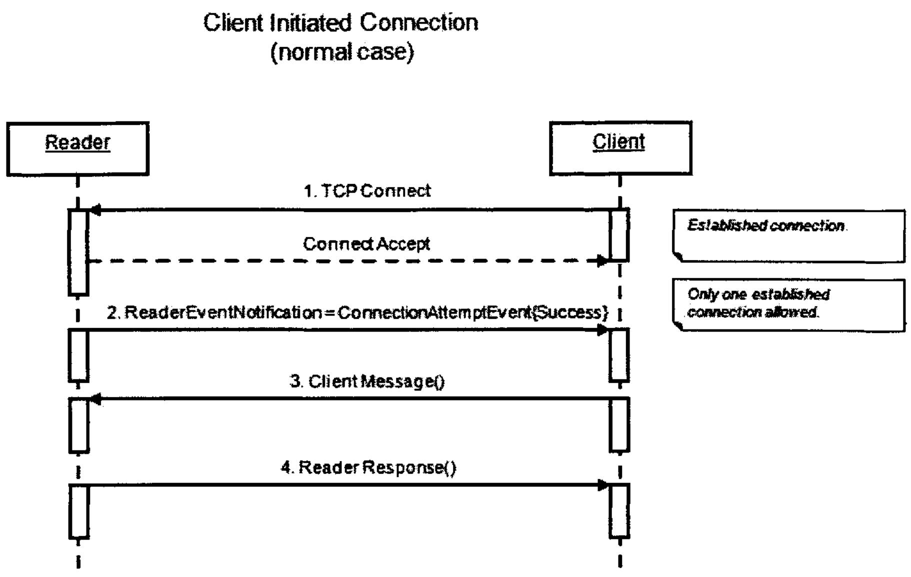

Client Initiated Connection
(normal case)
Reader
Client
1. TCP Connect
Connect Accept
Established connection.
Only one established connection allowed.
2. ReaderEventNotification=ConnectionAttemptEvent{Success}
3. Client Message()
4. Reader Response()

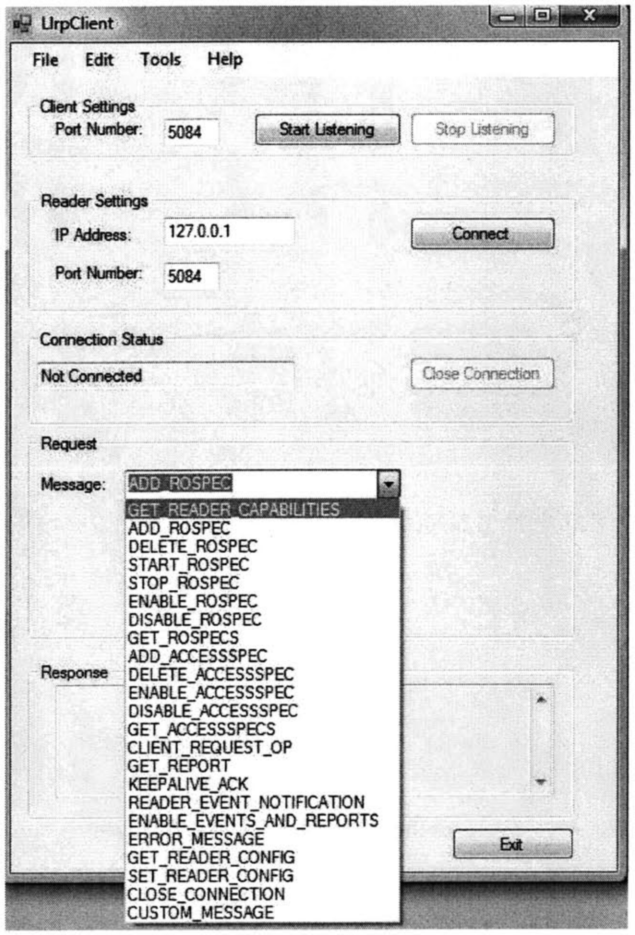
Figure 4-3: Client Initiated Connection

Messages are sent to the Reader asynchronously and, therefore, do not block the form controls while waiting for the Reader to respond. It is possible to send multiple Messages before receiving a response from the Reader. In this implementation Reader responses are simply displayed in the Response box as shown in Figure 4-7. There is no logic to keep track of outstanding Client Messages and no checks that the Message IDs of the received responses match the ID of the request.

4.3 Connection Termination

The LLRP connection can be terminated both by the Client and the Reader. A Reader terminates the connection by sending a status Message with a `ConnectionCloseEvent` Parameter. Following the status Message the Reader immediately closes the TCP connection without waiting for a Client response. In turn, the Client handles the connection termination and performs any cleanup operations if necessary.

The Client can also close the connection by sending a `CLOSE_CONNECTION` Message. The Client GUI includes a *Close Connection* button which sends a `CLOSE_CONNECTION`



Client Settings
Port Number: 5084

Reader Settings
IP Address: 127.0.0.1
Port Number: 5084

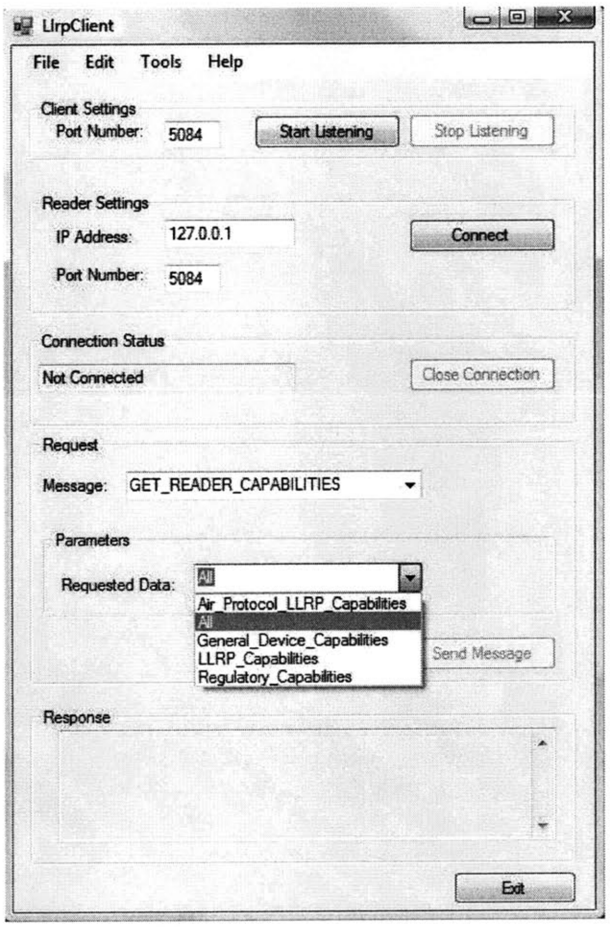
Connection Status
Not Connected

Request
Message: ADD_ROSPEC

Response

Figure 4-4: Message selection using drop-down menu

Message to the Reader. The Reader acknowledges the request to close the connection by sending a `CLOSE_CONNECTION_RESPONSE` Message and immediately closes the TCP connection. If the Reader fails to respond with a `CLOSE_CONNECTION_RESPONSE` Message within a certain time interval then the Client forces a connection termination.


LlrpClient

File Edit Tools Help

Client Settings
Port Number: 5084

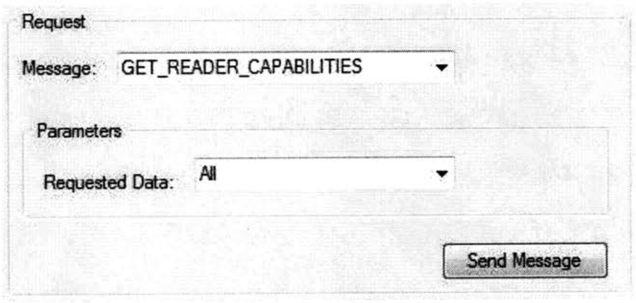
Reader Settings
IP Address: 127.0.0.1
Port Number: 5084

Connection Status
Not Connected

Request
Message: GET_READER_CAPABILITIES

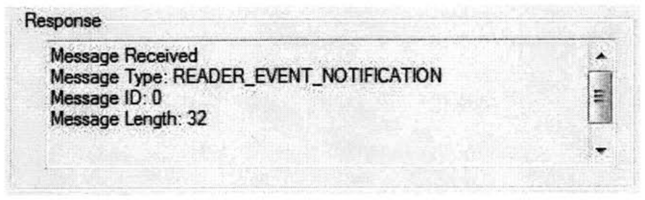
Parameters
Requested Data: All

Figure 4-5: Parameter Configuration


Request
Message: GET_READER_CAPABILITIES

Parameters
Requested Data: All

Figure 4-6: Configured Message can be send to Reader


Response

```
Message Received
Message Type: READER_EVENT_NOTIFICATION
Message ID: 0
Message Length: 32
```

Figure 4-7: Reader response displayed in Response box

Chapter 5

Discussion and Future Work

5.1 Discussion

5.1.1 Advantages of Object-Oriented implementation

Developing an object-oriented implementation of LLRP was the biggest design decision of this project. It might seem like an unusual choice for an implementation of a low level protocol but it offers a number of advantages over a low level implementation.

Ease of implementation Client logic is implemented using the object model which makes it easier to develop rich and robust Client applications. The low level details of the LLRP interface are abstracted away, eliminating in this way one of the major complexities of LLRP.

Use of enumerations instead of Integers Some Parameters are represented by integer values in the binary format but only have a small range of possible values. In some cases it is better to abstract away the integer values and use enumerations. Figure 5-1 the use of enumerations in the case of the ProtocolID Parameter. The Client does not need to know the underlying value which represents the *EPCGlobalClass1Gen2* and simply use `ProtocolID.EPCGlobalClass1Gen2`.

```
public enum ProtocolID : byte
{
  UnspecifiedAirProtocol = 0,
  EPCGlobalClass1Gen2 = 1
}
```

Figure 5-1: ProtocolID enumeration

Use of Interfaces The use of interfaces allows Parameters to be grouped together. For example a `SET_READER_CONFIG` Message includes a list of `IConfigSetParameters`. Parameters which can be used for configuring Reader operation implement the `IConfigSetParameter` Parameter. In this way it is ensured that only valid Parameters are passed in the configuration data list without the `SET_READER_CONFIG` class explicitly knowing which Parameters are valid.

Error Checking This implementation provides minimal error checking to facilitate faster development and testing. Messages and Parameters can only include valid fields and sub-Parameters but there are no checks to ensure that mandatory Parameters are present. Additionally, during deserialization there are checks to ensure that received Messages are well formed but it is possible to construct a Message that is missing mandatory Parameters. It is easy, however, to include additional checks for Parameter validity and increase the robustness of the application.

Parameter Range Checking The range of values for Parameters of basic types is dictated by the Parameter type. In some cases, however, the specification dictates a shorter range of values or excludes certain values or combinations of values. Even though Parameter range checks are not present in this implementation, they could easily be added on Parameter construction.

5.2 Future Work

In this section I propose some extensions to this LLRP implementation that have not yet been implemented and would ideally be implemented in the future.

5.2.1 Message Templates

The object model simplifies the construction of Messages and Parameters by providing object constructors instead of working directly with byte arrays. However, constructing a deeply nested object can still be tedious since every Parameter and sub-Parameter has to be constructed and assigned its value. To further simplify this process, the toolkit could provide Message Templates which would allow for common Messages to be constructed without the need to explicitly construct Parameter objects.

It could be possible to provide quick Message construction for all Messages by providing a default constructor for each Parameter, which would construct all sub-Parameters and assigned them default values. The Client would only need to call the Message constructor to create a Message object and it can use the access methods to modify Parameter values if needed.

5.2.2 Graphical User Interface

The Graphical LLRP Client implementation described earlier is a very simple one with limited functionality. The goal of this implementation was simply to illustrate how the LLRP library can be used to develop a Client application. As a future extension this GUI could be extend to provide much richer functionality. Message construction is currently limited to a few Message types which do no have deeply nested Parameters. Developing a UI to handle deeply nested Parameters can be a challenging task and might require complete redesign of the current interface.

Additional functionality could include handling muliple Readers and providing ways to automate Reader control. Readers can be controlled to send status reports either periodically or triggered by some event. The Client could provide logic to automatically process Reader responses and perform different actions depending on Reader status.

5.2.3 LLRP Schema

The current LLRP specification does not provide a schema for LLRP Messages and Parameters. Providing a schema would offer many advantages and assist in creating a more robust LLRP implementation. The schema should define all the Message and Parameter Types and specify the fields and sub-Parameters associated with each Message and Parameter. The schema should also indicate whether a Parameter is optional or if it can appear multiple times in a Message.

Using the LLRP schema it would be possible to create a module that would automatically generate all Message and Parameter classes, since the class structure is similar across Messages and Parameters. This would make the implementation of an LLRP library simpler and more robust. Possible future changes in the LLRP spec would be reflected in the schema and would not require any changes in the code. Moreover, the auto-generation module could be modified to provide Parameter Range checking and improved error checking without the need to tediously go through each Message and Parameter class.

5.2.4 TLS

LLRP connections could be secured by using the TLS protocol [1], which provides privacy and data integrity between two authenticated applications. In order to use TLS both Client and Reader must support its use. As a future extension this implementation could be extended to add support for TLS.

5.2.5 XML Encoding

A future extension for LLRP could be the addition of an XML encoding, which could be used instead of the binary format. Figure 5-2 illustrates how an LLRP Message could be encoded in XML format. The header parts and Message value each have their own XML nodes. The Message value would form a nested XML tree, with Parameters nested at an arbitrarily deep level.

The XML encoding is perhaps beyond the original scope of LLRP and might be

```
<Message>
  <Version>1.0</Version>
  <MessageType>ADD_ACCESS_SPEC</MessageType>
  <MessageID>12345</MessageID>
  <MessageValue>
    <Parameter>
      <ParameterName>accessSpec</ParameterType>
      <ParameterType>AccessSpec</ParameterType>
      <ParameterValue>
        <Parameter>
           <ParameterName>antennaId</ParameterType>
           <ParameterType>uint</ParameterType>
           <ParameterValue>1</ParameterValue>
        </Parameter>
        <Parameter>
          ...
        </Parameter>

        ...

      </ParameterValue>
    </Parameter>
  </MessageValue>
</Message>
```

Figure 5-2: XML Encoding for LLRP Message

harder to be adopted as it would require Readers to process XML documents. It could offer, however, a number of advantages compared to the binary format. Parsing XML would be much easier than parsing the binary format. The use of XPath or XQuery would also allow fast searching through an XML encoded Message and could allow Messages to be pre-processed before converting the XML into the object model. The XML Messages could be converted to Binary XML before being transported between Clients and Readers in order to limit the transportation overhead of XML.

Chapter 6

Conclusions

This Master's Thesis describes the design and implementation of an object-oriented LLRP library. LLRP is a recently released protocol concerned with the formats and procedures of communication between RFID Readers and Clients. The lack of standards in current RFID networks was the main motivation behind the development of the LLRP protocol. As RFID technology continues to be more widely adopted, it becomes imperative to standardize the way information is exchanged between RFID Readers and enterprise networks. A standard interface will allow RFID systems to be designed without dependence on vendor proprietary interfaces and will simplify their implementation. It is envisioned that LLRP will soon be adopted as the standard Reader - Client interface.

Although LLRP is well received by the RFID Reader industry, it is not yet widely adopted. The protocol is message-oriented and provides a high level of Reader control while ensuring interoperability, allowing the addition of vendor-specific extensions and provisioning for support of future air protocols. Consequently, LLRP is more complex than a protocol designed for more specific operation, making its adoption a challenging task.

The implementation described in this Thesis was oriented towards reducing the complexity of LLRP, which primarily lies in the binary nested format used for representing LLRP Messages and Parameters. A nested object model is used to represent all the Messages and Parameters defined in the LLRP specification. A Serialization module converts Message and Parameter objects to the binary format and vice versa, abstracting in this way the low level details of the LLRP interface.

The use of an object-oriented LLRP library simplifies the implementation of Client logic and makes it possible to develop rich Client applications. Working with objects instead of directly manipulating the binary format can make Client applications more robust and less error prone. It is also possible to check for Parameter validity and ensure that Parameters have values within the valid range. As showcased by the prototype implementation of a Graphical LLRP Client, this LLRP library can be a useful toolkit in developing LLRP applications, without requiring the investment of too much time and effort in dealing with the complexities of the LLRP interface.

Appendix A

Design Details

A.1 Casing Conventions

The following conventions are used for naming the various classes of the LLRP library:

ALL_CAPS_UNDERSCORE type is used for LLRP message names, e.g. GET_READER_CAPABILITIES

Camel casing is used for LLRP Parameter and data field names. e.g. ROSpec

Interfaces use camel casing and are prefixed with an I, e.g. IConfigGetParameter

A.2 Namespaces

LLRP.Messages.ReaderDeviceCapabilities

Messages that query Reader capabilities.

- GET_READER_CAPABILITIES
- GET_READER_CAPABILITIES_RESPONSE

LLRP.Messages.ReaderOperation

Messages that control the Readers air protocol inventory and RF operations.

- ADD_ROSPEC
- ADD_ROSPEC_RESPONSE

- DELETE_ROSPEC
- DELETE_ROSPEC_RESPONSE
- START_ROSPEC
- START_ROSPEC_RESPONSE
- STOP_ROSPEC
- STOP_ROSPEC_RESPONSE
- ENABLE_ROSPEC
- ENABLE_ROSPEC_RESPONSE
- DISABLE_ROSPEC
- DISABLE_ROSPEC_RESPONSE
- GET_ROSPECS
- GET_ROSPECS_RESPONSE

LLRP.Messages.AccessOperation

Messages that control the tag access operations performed by the Reader.

- ADD_ACCESSSPEC
- ADD_ACCESSSPEC_RESPONSE
- DELETE_ACCESSSPEC
- DELETE_ACCESSSPEC_RESPONSE
- ENABLE_ACCESSSPEC
- ENABLE_ACCESSSPEC_RESPONSE
- DISABLE_ACCESSSPEC

- DISABLE_ACCESSSPEC_RESPONSE
- GET_ACCESSSPECS
- GET_ACCESSSPECS_RESPONSE
- CLIENT_REQUEST_OP
- CLIENT_REQUEST_OP_RESPONSE

LLRP.Messages.ReaderDeviceConfiguration

Messages that query/set Reader configuration, and close LLRP connection.

- GET_READER_CONFIG
- GET_READER_CONFIG_RESPONSE
- SET_READER_CONFIG
- SET_READER_CONFIG_RESPONSE
- CLOSE_CONNECTION
- CLOSE_CONNECTION_RESPONSE

LLRP.Messages.ReportsNotificationsKeepalives

Messages that carry different reports from the Reader to the Client. Reports include Reader device status, tag data, RF analysis report.

- GET_REPORT
- RO_ACCESS_REPORT
- READER_EVENT_NOTIFICATION
- KEEPALIVE
- KEEPALIVE_ACK
- ENABLE_EVENTS_AND_REPORTS

LLRP.Messages.CustomExtension

Messages that contain vendor defined content.

- CUSTOM_MESSAGE

LLRP.Messages.Errors

Generic Error Messages.

- ERROR_MESSAGE

LLRP.Parameters.ReaderOperation

Parameters for Messages that control the Readers air protocol inventory and RF operations.

LLRP.Parameters.AccessOperation

Parameters for Messages that control the tag access operations performed by the Reader.

LLRP.Parameters.ReaderDeviceConfiguration

Parameters for Messages that query/set Reader configuration, and close LLRP connection.

LLRP.Parameters.ReportsNotificationsKeepalives

Parameters for Messages that carry different reports from the Reader to the Client.

LLRP.Parameters.ReportsNotificationsKeepalives.ReaderEventNotificationData

Parameters for Messages that carry different event notification reports from the Reader to the Client.

LLRP.Parameters.ReportsNotificationsKeepalives.TagReportData

Parameters for Messages that carry different tag data reports from the Reader to the Client.

LLRP.Parameters.ReaderDeviceCapabilities

Parameters for Messages that query Reader capabilities.

LLRP.Parameters.ReaderDeviceCapabilities.GeneralDeviceCapabilities

Parameters for Messages that query general device capabilities of a Reader.

LLRP.Parameters.ReaderDeviceCapabilities.RegulatoryCapabilities

Parameters for Messages that query regulatory device capabilities of a Reader.

LLRP.Parameters.General

General Parameters.

LLRP.Parameters.AirProtocolSpecific.C1G2

Parameters for the C1G2 air protocol.

LLRP.Parameters.CustomExtension

Parameters for Messages that contain vendor defined content.

LLRP.Parameters.Errors

Parameters for generic error Messages

A.3 Class Diagrams

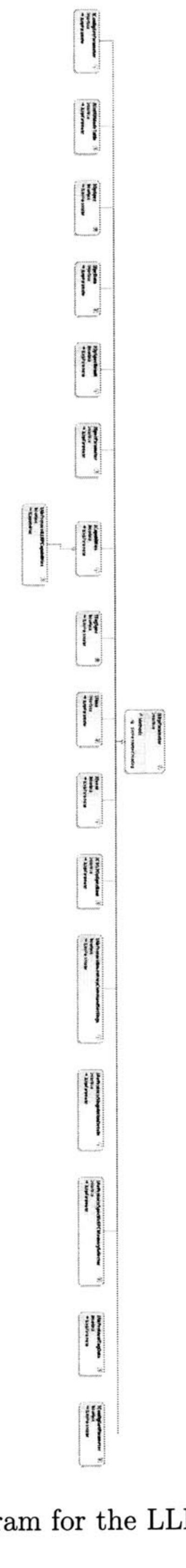


Figure A-1: Class Diagram for the LLRP Parameter Interfaces

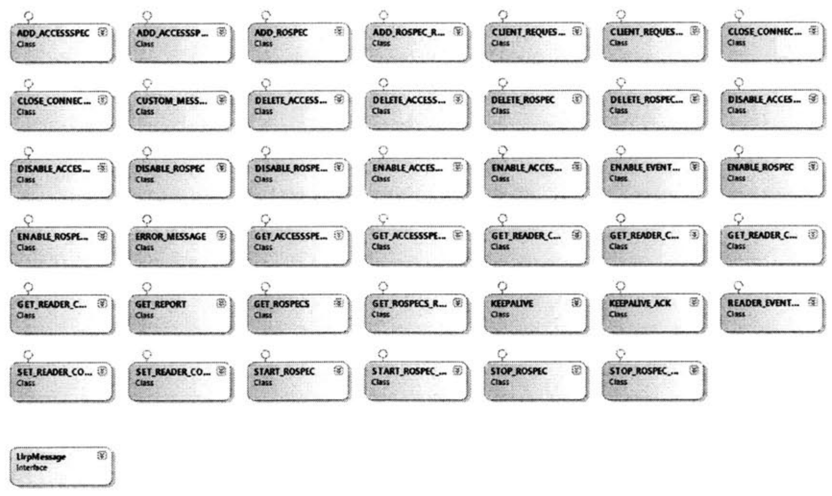


Figure A-2: Class Diagram for LLRP Messages

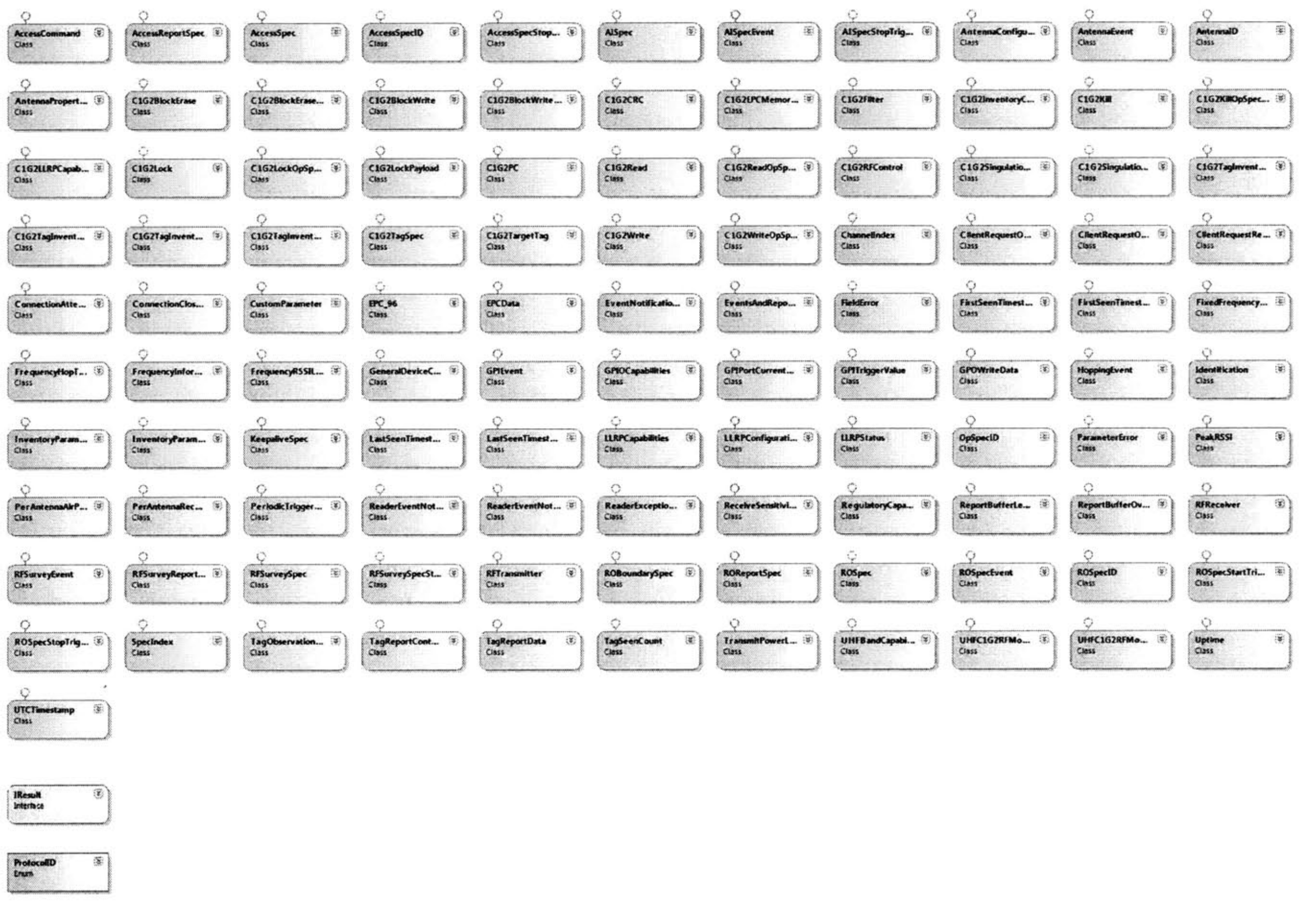


Figure A-3: Class Diagram for LLRP Parameters

Appendix B

Serialization Methods

B.1 Objects to byte[]

```
/// <summary>
/// Returns the specified list of parameters as a byte array.
/// Each parameter on the list is recursively converted into a
/// byte array with all resulting byte arrays are merged into a single
/// byte array.
/// </summary>
/// <param name=p̈arameterList">
/// A list of objects.
/// List elements must be of one of the following types:
/// 1. Native type
/// 2. LLRPParameter
/// 3. List of LLRPParameters
/// </param>
/// <returns></returns>
  public static byte[] GetBytes(List<object> parameterList)
  {
      if (parameterList == null)
        return new byte[0];

      int byteArrayLength = 0;
```

```
        List<byte[]> byteArrayList = new List<byte[]>(parameterList.Count);

        // Go through each element in the list, convert it to a byte array
        // and insert byte arrays in an auxiliary list
        foreach (object parameter in parameterList)
        {
          if (parameter != null)
          {
            byte[] encoding = GetBytes(parameter);
            byteArrayList.Add(encoding);
            byteArrayLength += encoding.Length;
          }
        }

        // Go through the auxiliary list containing all the byte arrays, and
        // merge them into a single byte array
        byte[] result = new byte[byteArrayLength];
        int offset = 0;
        foreach (byte[] b in byteArrayList)
        {
           b.CopyTo(result, offset);
           offset += b.Length;
        }

        return result;
}
```

B.1.1 Integer Types

```
/// <summary>
/// Returns the specified 32-bit signed integer value as an array of bytes.
/// The integer is encoded in network byte order with the most significant
/// byte of the integer first (Big-Endian).
/// </summary>
/// <param name=v̈alue¨>The number to convert</param>
/// <returns>A byte array of length 4</returns>
public static byte[] GetBytes(int value)
{
  byte[] result = new byte[INT_BYTE_LENGTH];

  for (int i = 0; i < INT_BYTE_LENGTH; i++)
    result[INT_BYTE_LENGTH - 1 - i] = (byte)(value >> (i * 8));

  return result;
}
```

B.1.2 Strings

```
/// <summary>
/// Returns the specified string value as an array of bytes.
/// The string is encoded with UTF-8 encoding.
/// The first two bytes of the array specify the length of
/// the UTF-8 encoded string in bytes.
/// </summary>
/// <param name=v̈alue¨>The string valye</param>
/// <returns>A byte array of variable length</returns>
public static byte[] GetBytes(string value)
{
  // Get UTF-8 encoding
  byte[] utfEncoding = UTF8Encoding.UTF8.GetBytes(value);

  // Get length of string in bytes
  ushort byteLength = (ushort)utfEncoding.Length;

  byte[] result = new byte[USHORT_BYTE_LENGTH + byteLength];

  InsertBytes(result, 0, byteLength);
  utfEncoding.CopyTo(result, USHORT_BYTE_LENGTH);
  return result;
}
```

B.1.3 Booleans

```
/// <summary>
/// Returns the specified Boolean value as an array of bytes.
/// </summary>
/// <param name=v̈alue¨>A Boolean value</param>
/// <returns>A byte array of length 1</returns>
public static byte[] GetBytes(bool value)
{
  // Insert bool value in a byte array of length 1
  byte[] result = new byte[BYTE_LENGTH];

  // If value is true set the first bit of the byte to 1
  if (value)
    result[0] |= 0x1;

  return result;
}
```

B.2 Byte[] to Objects

B.2.1 Integer Types

```
/// <summary>
/// Returns a 32-bit signed integer converted from four bytes
/// at a specified position in a byte array.
/// </summary>
/// <param name=b̈uffer¨>An array of bytes.</param>
/// <param name=s̈tartIndex¨>The starting position within buffer.</param>
/// <exception cref=ÄrgumentOutOfRangeException¨>
/// startIndex is less than zero or greater than the length of value minus 4.
/// </exception>
/// <exception cref=ÄrgumentNullException¨>Buffer is null</exception>
/// <returns> A 32-bit signed integer formed by four bytes beginning at startIndex.</returns>
public static int ToInt(byte[] buffer, ref int startIndex)
{
  if (buffer == null)
    throw new ArgumentNullException(S̈pecified Buffer is null)̈;
  if (startIndex < 0)
    throw new ArgumentOutOfRangeException(s̈tartIndex is less than 0)̈;
  if (startIndex + INT_BYTE_LENGTH > buffer.Length)
    throw new ArgumentOutOfRangeException(s̈tartIndex is greater than the length of value minus 4)̈;

  int endIndex = startIndex + INT_BYTE_LENGTH - 1;
  int result = 0;
  for (int i = 0; i < INT_BYTE_LENGTH; i++)
  {
    result |= (int)buffer[endIndex - i] << (i * 8);
  }

  startIndex += INT_BYTE_LENGTH;

  return result;
}
```

B.2.2 Strings

```
/// <summary>
/// Returns a string converted from variable length UTF-8 encoded
/// byte array, starting at the specified position in the byte array.
/// </summary>
/// <param name=b̈uffer¨>An array of bytes.</param>
/// <param name=s̈tartIndex¨>The starting position within buffer.</param>
/// <exception cref=ÄrgumentOutOfRangeException¨>
/// startIndex is less than zero or startIndex and byteLength
/// do not denote a valid range in bytes.
/// </exception>
/// <exception cref=ÄrgumentNullException¨>Buffer is null</exception>
/// <returns> A string formed by a variable length array, starting at startIndex.</returns>
public static string ToString(byte[] buffer, ref int startIndex)
{
  if (buffer == null)
    throw new ArgumentNullException(S̈pecified Buffer is null)̈;

  ushort byteLength = ToUShort(buffer, ref startIndex);
  string result = UTF8Encoding.UTF8.GetString(buffer, startIndex, byteLength);

  startIndex += byteLength;
  return result;
}
```

B.2.3 Booleans

```
/// <summary>
/// Returns a bool converted from a byte at a specified position in a byte array.
/// Returns true if the first bit of the byte is set to 1, false otherwise
/// </summary>
/// <param name=b̈uffer¨>An array of bytes.</param>
/// <param name=s̈tartIndex¨>The starting position within buffer.</param>
/// <exception cref=ÄrgumentOutOfRangeException¨>
/// startIndex is less than zero or greater than the length of value minus 1.
/// </exception>
/// <exception cref=ÄrgumentNullException¨>Buffer is null</exception>
/// <returns> A bool formed by a byte beginning at startIndex.</returns>
public static bool ToBool(byte[] buffer, ref int startIndex)
```

```
{
  if (buffer == null)
    throw new ArgumentNullException(Špecified Buffer is null)̈;
  if (startIndex < 0)
    throw new ArgumentOutOfRangeException(štartIndex is less than 0)̈;
  if (startIndex + BYTE_LENGTH > buffer.Length)
    throw new ArgumentOutOfRangeException(štartIndex is greater than the length of value minus 1)̈;

  byte firstBitValue = GetBits(buffer[startIndex++], 0, 1);
  if (firstBitValue == 1)
    return true;

  return false;
}
```

B.2.4 LLRP Parameters

```
/// <summary>
/// Returns an object of type T converted from a variable number of bytes
/// depending on object type, starting at the specified position in
/// a byte array
/// </summary>
/// <typeparam name=T̈¨>
/// The object type. The type must be implementing the ILlrpParameter interface.
/// </typeparam>
/// <param name=b̈uffer¨>A byte array.</param>
/// <param name=štartIndex¨>The starting position within buffer.</param>
/// <returns>
/// An object of type T formed by a variable number of bytes starting at startIndex
/// </returns>
public static T ToILlrpParameter<T>(byte[] buffer, ref int startIndex)
  where T : ILlrpParameter
{
  T expectedParameter = default(T);
```

```
if (startIndex < buffer.Length)
{
  // Check first bit to determine parameter encoding
  // All TLV-encoded parameters have a 0 in bit 0 of the header
  // All TV-encoded parameters have a 1 in bit 0 of the header
  byte firstByte = buffer[startIndex];
  byte firstBit = GetBits(firstByte, 0, 1);

  if (firstBit == 0)
  {
    //TLV Encoding
    TLVParameterEncoding parameterEncoding = TLVParameterEncoding.GetTLVParameter(buffer, startIndex
    ILlrpParameter parameter = LlrpFactory.GetILlrpParameter(parameterEncoding);

    if (typeof(T).IsAssignableFrom(parameter.GetType()))
    {
      expectedParameter = (T)parameter;
      startIndex += parameterEncoding.ParameterLength;
    }
  }
  else
  {
    //TV Encoding
    TVParameterEncoding parameterEncoding = TVParameterEncoding.GetTVParameter(buffer, startIndex
    ILlrpParameter parameter = LlrpFactory.GetILlrpParameter(parameterEncoding);
    if (typeof(T).IsAssignableFrom(parameter.GetType()))
    {
      expectedParameter = (T)parameter;
      startIndex += parameterEncoding.ParameterLength;
    }
  }


}
return expectedParameter;
}
```

B.2.5 Parameter Lists

```
/// <summary>
/// Returns a list of objects of type T converted from a variable number of bytes
/// depending on object type, starting at the specified position in
/// a byte array
/// </summary>
/// <typeparam name=T̈¨>
/// The object type. The type must be implementing the LLRPParameter interface.
/// </typeparam>
/// <param name=b̈uffer¨>A byte array.</param>
/// <param name=s̈tartIndex¨>The starting position within buffer.</param>
/// <returns>
/// A List of objects of type T formed by a variable number of bytes starting at startIndex
/// </returns>
public static List<T> ToILlrpParameterList<T>(byte[] buffer, ref int startIndex)
  where T : ILlrpParameter
{
  List<T> parameterList = new List<T>();

  while (startIndex < buffer.Length)
  {
    T parameter = ToILlrpParameter<T>(buffer, ref startIndex);

    if (parameter != null)
      parameterList.Add(parameter);
    else
      break;
    }

  return parameterList;
}
```

Bibliography

[1] T. Dierks and C. Allen. *The TLS Protocol Version 1.0.* United States, 1999.

[2] EPCGlobal. *The Application Level Events (ALE) Specification, Version 1.0.* EPCGlobal Standard Specification, September 2005.

[3] EPCGlobal. *Class 1 Generation 2 UHF Air Interface Protocol Standard Version 1.0.9.* EPCGlobal Standard Specification, January 2005.

[4] EPCGlobal. *Object Naming Service (ONS) Version 1.0.* EPCGlobal Standard Specification, October 2005.

[5] EPCGlobal. *The EPCGlobal Architecture Framework.* EPCGlobal Standard Specification, July 2006.

[6] EPCGlobal. *EPC Information Services (EPCIS) Version 1.0.* EPCGlobal Standard Specification, April 2007.

[7] EPCGlobal. *Low Level Reader Protocol (LLRP) Version 1.0.* EPCGlobal Standard Specification, April 2007.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[9] Sanjay Sarma, David Brock, and Daniel Engels. Radio frequency identification and the electronic product code. *IEEE Micro*, 21(6):50–54, 2001.

[10] F. Yergeau. *UTF-8, a transformation format of ISO 10646.* United States, 1998.