LTK Programmers Guide

Version 7.6

Copyright © 2012 - 2021 Impinj, Inc. All rights reserved

http://www.impinj.com

Impinj, Octane, Speedway, xSpan and xArray are either registered trademarks or trademarks of Impinj, Inc. Visit www.impinj.com/trademarks for additional information about Impinj trademarks.

Contents

1 Introduction **3**

1.1 Purpose of Guide . . . 3

1.2 Intended Audience . . . 3

1.3 Overview of Contents . . . 3

1.4 Document Conventions and Terms . . . 4

1.4.1 Document Terms . . . 5

1.5 Technical Support . . . 5

1.6 Other Reference Material . . . 6

1.6.1 LLRP Standards Document . . . 6

1.6.2 Octane LLRP Reference Guide . . . 7

1.6.3 Speedway Embedded Developers' Guide . . . 7

1.6.4 Language-Specific API Reference Documentation . . . 7

1.6.5 www.LLRP.org . . . 7

1.6.6 LLRP Toolkit Project on Source-Forge . . . 7

1.6.7 Impinj LTK Extension Files . . . 8

1.6.8 Impinj Low Level User Data Support Application Note . . . 8

2 LLRP Overview **9**

2.1 Origin and Purpose . . . 9

2.2 Connection Details . . . 9

2.3 Declarative Specifications . . . 10

2.4 Messages, Parameters, and Fields . . . 11

2.4.1 Messages . . . 11

2.4.2 Fields . . . 12

2.4.3 Parameters . . . 13

2.5 Introduction to LLRP Specs . . . 13

2.5.1 Reader Operation Specs (ROSpecs) . . . 14

2.5.2 Access Specs . . . 16
2.6 A typical LLRP Exchange . . . 18
2.7 Vendor Extensions to LLRP . . . 20
2.8 Impinj LLRP Extensions . . . 21

3 LTK Overview 22
3.1 Origin and Purpose . . . 22
3.2 LTK versus LLRP . . . 22
3.3 LTK Definition Files . . . 23
3.4 LTK Vendor Extensions . . . 25
3.5 LTK-XML . . . 25

4 Getting Started with the Impinj LTK 28
4.1 Overview . . . 28
4.1.1 Impinj Supported Platforms and Languages . . . 28
4.1.2 CHANGES.TXT . . . 29
4.1.3 Version Names . . . 29
4.2 Obtaining the LTK . . . 29
4.3 LTK Required Tools/Environment . . . 29
4.4 Installing and Running the LTK Samples . . . 30
4.4.1 LTKC . . . 30
4.4.2 LTKCPP . . . 31
4.4.3 LTKNET . . . 35
4.4.4 LTKJava . . . 40

5 LTK Language Details 52
5.1 LTKCPP (LTKC) . . . 52
5.1.1 Connection Class . . . 52
5.1.2 Element Class . . . 53
5.1.3 Fields, Subparameters, and Accessor Functions . . . 54
5.1.4 Why It Is Best to Use the Accessor Functions . . . 56

5.1.5 How to Determine the Type of an Element . . . 57
5.1.6 Practical Information about Building Messages in LTKCPP . . . 58
5.1.7 Error Reporting (ErrorDetails Class) . . . 60
5.1.8 Extensions . . . 61
5.2 LTKNET . . . 63
5.2.1 LLRPClient . . . 63
5.2.2 Impinj Installer Class . . . 64
5.2.3 LLRP Message Classes . . . 65
5.2.4 Handling ROSpecs . . . 66
5.2.5 Handling AccessSpecs . . . 67
5.3 LTKJAVA . . . 69
5.3.1 Receiving LLRP Messages . . . 69
5.3.2 Connections . . . 70
5.3.3 Sending and Receiving Messages . . . 70
5.3.4 LLRPMessage and LLRPParameter . . . 71
5.3.5 Building LLRP Messages . . . 71
5.3.6 Custom Extensions . . . 73
5.3.7 Logging . . . 74
5.3.8 Summary . . . 74

6 Adding the LTK into your programs 75
6.1 Compiling and Linking . . . 75
6.2 Initializing the library . . . 79
6.3 Connecting to the Reader . . . 80
6.3.1 LTKCPP . . . 80
6.3.2 LTKNET . . . 84
6.3.3 LTKJAVA . . . 84
6.4 Enabling the Impinj Extensions . . . 85
6.4.1 LTKCPP . . . 85
6.4.2 LTKNET . . . 87

6.4.3 LTKJAVA . . . 88
6.5 Debugging with LTK . . . 89
6.5.1 LTKC . . . 90
6.5.2 LTKCPP . . . 90
6.5.3 LTKNET . . . 91
6.5.4 LTKJava . . . 91

7 Creating a Simple Impinj LTK Application 93
7.1 Application Flow . . . 93
7.2 Setup . . . 95
7.3 Factory Defaults for the Reader . . . 95
7.3.1 LTKCPP . . . 96
7.3.2 LTKNET . . . 97
7.3.3 LTKJAVA . . . 99
7.4 Add an ROSpec . . . 100
7.4.1 LTKCPP . . . 102
7.4.2 LTKNET . . . 104
7.4.3 LTKJava . . . 106
7.5 Enable an ROSpec . . . 109
7.6 Start an ROSpec . . . 112
7.7 Handling Tag Report Data . . . 115
7.7.1 LTKCPP . . . 115
7.7.2 LTKNET . . . 117
7.7.3 LTKJava . . . 119

8 Impinj Octane LTK Application 123
8.1 Analysis . . . 123
8.2 Application Flow . . . 123
8.3 Impinj Capabilities/Getting Reader Capabilities . . . 124
8.4 Get Reader Configuration . . . 130

8.5 Impinj Configuration . . . 135
8.5.1 LTK-XML . . . 135
8.5.2 LTKNET . . . 137
8.5.3 LTKCPP . . . 139
8.5.4 LTKJAVA . . . 142

9 LLRP Access, Filtering, and Accumulation 145

9.1 Analysis . . . 145
9.2 Application Flow . . . 145
9.3 Report Accumulation . . . 146
9.3.1 LTKNET . . . 147
9.3.2 LTKCPP . . . 148
9.3.3 LTKJAVA . . . 150
9.4 Tag Filtering . . . 150
9.4.1 LTKNET . . . 150
9.4.2 LTKCPP . . . 151
9.4.3 LTKJAVA . . . 153
9.5 Tag Access . . . 154
9.5.1 LTKNET . . . 155
9.5.2 LTKCPP . . . 161
9.5.3 LTKJAVA . . . 164
9.6 Report Generation . . . 166
9.6.1 LTKNET . . . 166
9.6.2 LTKCPP . . . 167
9.6.3 LTKJAVA . . . 167

10 Impinj Octane LTK Tag Data 169

10.1 Analysis . . . 169
10.2 Application Flow . . . 169
10.3 Impinj Configuration . . . 170

10.3.1 LTK-XML 170
10.3.2 LTKNET 172
10.3.3 LTKCPP 172
10.3.4 LTKJAVA 174
10.4 Getting Tag Low-Level Data 174
10.4.1 LTKCPP 174
10.4.2 LTKNET 175
10.4.3 LTKJAVA 175
10.5 Computing Velocity 176

11 Using Monza 4 QT **177**
11.1 Adding Custom OpSpecs 177
11.1.1 LTKCPP 177
11.1.2 LTKNET 178
11.2 Retrieving Custom OpSpecResults 178
11.2.1 LTKCPP 178
11.2.2 LTKNET 179

12 xArray Gateway **180**
12.1 Example of ROSPEC for Location Role 180
12.2 Create a READER_CONFIGURATION for xArray Location Role in XML 181
12.3 Create an Event Handler for xArray in Java and C# 182

13 Revision History **186**

14 Notices **187**

1 Introduction

1.1 Purpose of Guide

This guide provides the experienced programmer with the information necessary to write RFID applications communicating with Impinj Speedway™ Readers ("Reader") via Octane™ LLRP. Information about using the LTK to write RFID applications that communicate with Impinj xArray Gateway ("xArray Gateway") is also included in section 12 in this document.

1.2 Intended Audience

This guide provides software and/or systems engineers with information for creating applications that will communicate with and control Readers via Low-Level Reader Protocol(LLRP). Prior programming knowledge in C/C++, Java, or C# is required. Familiarity with XML is desired. Prior knowledge of LLRP is helpful, but not required.

1.3 Overview of Contents

- Section 1 provides the introductory material needed to understand this document, including references and document formatting, as well as required and optional references.
- Section 2 provides information about LLRP.
- Section 3 provides the Impinj LLRP Toolkit (LTK) information.
- Section 4 provides detailed information about getting started with Impinj LTK programming.
- Section 5 covers LTK programming language details.
- Section 6 examines the details of adding LTK support to your applications, including communications between an application and either a Reader.
- Section 7 provides the process for creating a simple Impinj LTK application.
- Section 8 describes how to use Impinj Octane extensions via the Impinj LTK.
- Section 9 describes how to use the data filtering and access capabilities of LLRP via the Impinj LTK.
- Section 10 describes how to access the Impinj LLRP custom tag data elements via the Impinj LTK.
- Section 11 describes how to use Monza4 QT features to allow control of public and private areas of tag memory via the Impinj LTK.
- Section 12 describes how to use the the Impinj LTK to communicate with the xArray Gateway.
- Appendix A: Glossary provides a glossary of common terms used in this guide.

1.4 Document Conventions and Terms

Table 1.1 Document Style Conventions

Type	**Example**	**Style**
LLRP message	IMPINJ_ENABLE_EXTENSIONS	CAPS _UNDERSCORES
LLRP parameter	*AntennaConfiguration*	Italics Camel Case
LLRP field	*ResetToFactoryDefault*	Italics Camel Case
Enumerated field value	'Upon N Tags or End of AISpec'	'Single-quoted string'
File name	**'ImpinjDef.xml'**	'Single-quoted bold'
LTK function	***getLLRPStatus***	Bold italics case matches programming syntax
LTK class names	**CIMPINJ_TCS_RESPONSE**	Bold case matches programming syntax

Additional style usage:

- Important LLRP and LTK usage notes are shown in *italics* in a separate paragraph that begins with the words: **Impinj Best Practice!**
- Code examples are formatted in separate text boxes. The name of the toolkit is shown in bold at the top of the text box, and is not part of the compiled code example. When possible, color syntax highlighting the appropriate language is used. In many circumstances, there are several ways to complete a programming task in LTK. When such conditions arise, Impinj recommends best practices or recommended approaches to use when solving these problems. These sections are shown in code comments preceded by the following keywords: **Impinj Best Practice!**

LTKCPP

```
// a sample of what program text will look like
int x = 6;
int foo(int x)
{
   // **Impinj Best Practice!** - *sample best practices comment*
   return foo;
}
```

Figure 1.1 Documentation Code Sample

1.4.1 Document Terms

AccessSpec: Access Specification. This data element is passed to the Reader to describe a set of operations to perform on a tag. It includes a filter set that describes the tag population to which this rule applies. It also includes a list of read, write, lock, and kill commands to execute on each tag that matches the filter.

AISpec: Antenna Inventory Specification. An *ROSpec* contains a list of *AISpecs* that are executed in order. Each AISpec contains RF parameters, inventory parameters, and duration.

AntennaConfiguration: Each *AISpec* could contain one or more AntennaConfiguration parameters. These parameters describe the RF parameters (power, frequency, receive sensitivity) and Gen2 settings (mode, filters, session) to use during an AISpec execution.

Custom Extension: A mechanism of LLRP that allows vendors to add functionality beyond the standard behavior of LLRP.

EPC: Electronic Product Code. A unique identification code that is stored in the chip on an RFID tag as a product goes through the supply chain.

EPCglobal – EPCglobal is an organization that leads the development of industry-driven standards for the Electronic Product Code (EPC) that support the use of RFID.

FOV: Field-of-view. The angular extent of the observable world that is visible to a Reader at a given moment. This is typically related to antenna type, number, and position.

Impinj LTK: The LTK, which has been extended, compiled, tested, and packaged by Impinj for use with Readers.

LLRP: The EPCglobal Low Level Reader Protocol standard.

LTK: The llrp-toolkit, an open source LLRP library development project.

RO: Reader Operations. The group chartered within EPCglobal to define LLRP.

ROSpec: Reader Operation Specification. This data element is passed to the Reader to describe a bounded (start and end), triggered inventory operation.

STL: Standard Template Library.

XML: Extensible Markup Language. XML is the World Wide Web Consortium's (W3C) recommended standard for creating formats and sharing data on the Web.

1.5 Technical Support

For technical support issues related to LLRP, the Impinj LTK, or the use of Octane, Speedway, or xArray, visit `http://support.impinj.com`. The first time you visit the Impinj support portal, you must request a support account.

The support portal contains the Reader and xArray Gateway software downloads, Impinj LTK releases, additional code samples, and FAQs describing the use of Readers and xArray Gateways in common application scenarios.

1.6 Other Reference Material

1.6.1 LLRP Standards Document

The LLRP standard document located at http://www.epcglobalinc.org/standards/llrp is the normative reference for the LLRP protocol. If the general introduction to LLRP in Section 2 of this document does not provide sufficient detail, the best sources for additional information are the LLRP specification and other supporting materials provided by EPCglobal.

Sections 7-16 provide details for LLRP messages, parameters, and fields. These sections are a supplemental reference guide for Impinj LTK programmers. The relevant data has been extracted from the LLRP standard document into the Impinj LTK. To directly open the LLRP standard document, click the following link: http://www.gs1.org/gsmp/kc/epcglobal/llrp/llrp_1_0_1-standard-20070813.pdf.

- Section 6 provides the LLRP standard describes the theory of operation and provides the background information necessary to understand the LLRP control model and the high level features of LLRP. This section is very helpful to Impinj LTK programmers because it provides additional descriptions of the LLRP usage model.
- Sections 7 and 8 discuss the formatting conventions for the document and provide information on the logical structure of LLRP.
- Section 9 describes the messages, parameters, and fields required to discover Reader capabilities through LLRP.
- Section 10 describes the messages, parameters, and fields required to perform Reader operations (such as inventory) through LLRP.
- Section 11 describes the messages,parameters, and fields required to perform Reader access, including write/lock/kill through LLRP.
- Section 12 provides the messages, parameters, and fields required to perform Reader configuration through LLRP.
- Section 13 provides the messages, parameters, and fields required to collect report data (for example, tag reads) through LLRP.
- Section 14 describes the messages, parameters, and fields required to get error status through LLRP.
- Section 15 defines Gen2-specific messages, parameters, and fields.
- Section 16 describes the binary encoding of LLRP and should not be necessary for users of the LTK.

1.6.2 Octane LLRP Reference Guide

The *Octane LLRP* reference guide provides a summary to system architects for validating and understanding the standard LLRP features supported by Octane LLRP, as well as the unique Octane LLRP custom extensions that provide additional capabilities. This guide provides detailed information to developers who are planning to support Readers via LLRP.

Use *Octane LLRP* as a companion to this document. This document provides the structure for how to use the LTK for both standard LLRP and Octane LLRP extensions. The Octane LLRP guide provides the detailed information for each Impinj message, parameter, and field. The latest release of *Octane LLRP* is available on the Impinj support portal at `http://support.impinj.com`.

1.6.3 Speedway Embedded Developers' Guide

The *Speedway Embedded Developers' Guide* provides instructions for compiling and loading applications that run on the Speedway Reader platform.

The latest release of the *Speedway Embedded Developers' Guide* is available on the Impinj support portal at `http://support.impinj.com`.

1.6.4 Language-Specific API Reference Documentation

Some of the LTK implementations have built-in documentation on the API language specifics. View the individual LTK downloads on `http://support.impinj.com` for more information.

1.6.5 www.LLRP.org

Read more about LLRP at `http://www.llrp.org`, which provides general information on the open source LLRP toolkit effort. It includes information about companies and organizations that use the LTK, and information about open source tools and documentation to accompany LTK or LLRP. This site does not contain any information about the Impinj-specific extensions to LLRP or LTK.

1.6.6 LLRP Toolkit Project on Source-Forge

The Impinj LTK is heavily based on the open source project llrp-toolkit hosted at source-forge. You can access the project at `http://www.sourceforge.net/projects/llrp-toolkit`.

Impinj is a committed contributor to the llrp-toolkit project. We submit all bug fixes and enhancements that benefit the community llrp-toolkit project. Because of the disparity in Impinj and llrp-toolkit release cycles, individual versions of the Impinj LTK may not match the open source behavior exactly.

The LTK is supported in several languages, including languages not formally released by Impinj, such as Perl.

In order to build LLRP libraries with support for Impinj Octane extensions, the Impinj LTK extension xml files are required. Impinj extensions also allow LTK-XML based applications to validate their packet descriptions. The latest release of the Impinj LTK Extension files is available on the Impinj support portal at http://support.impinj.com.

1.6.7 Impinj LTK Extension Files

In order to build the open source llrp-toolkit with support for Impinj extensions, the Impinj LTK extension files are required. These files allow the toolkit to automatically generate support for Octane LLRP extensions. Impinj extensions also allow LTK-XML based applications to validate their packet descriptions. The latest release of the Impinj LTK Extension files is available on the Impinj support portal at http://support.impinj.com.

1.6.8 Impinj Low Level User Data Support Application Note

Starting with Octane 4.4, the Reader Gateway supports RF phase and RSSI reporting through custom LLRP extensions. The *ImpinjTagReportContentSelector* parameter allows the operator to configure additional parameters to be reported via the *TagReportData* parameter including the *ImpinjRFPhaseAngle* and *ImpinjPeakRSSI* parameters. For a complete description of how to enable the low level user data feature for the Reader, refer to the *Octane LLRP Reference Guide* and the application notes, titled *Low Level User Data Support*. Both documents are available on the Impinj support portal at http://support.impinj.com.

2 LLRP Overview

This section provides an introduction to LLRP. If you are already familiar with the LLRP standards, you can continue to section 3. For a more complete description of LLRP, refer to the EPCglobal standards documentation referenced in Section 1.6.1.

2.1 Origin and Purpose

In April 2007, EPCglobal announced ratification of the Low-Level Reader Protocol (LLRP) standard, a specification for the network interface between the Reader and its controlling application. Now that the tag and reader radio frequency (RF) air interface protocol with UHF Gen 2 has been standardized, the LLRP specification was the practical, logical next step to facilitate the adoption of RFID. Impinj recognized the importance of both standardization and collaboration with other companies as a vehicle for moving RFID toward more widespread adoption. Similar to Impinj's role in the UHF Gen 2 air protocol standard definition, Impinj assumed a leadership role, dedicating important internal resources who worked closely with other companies to define the standard and to work toward open source programming support.

The device that operates an RFID reader can vary from a software application, to middleware running on server hardware, to programmable logic controllers. Even though the ultimate end-business application that these clients support might be different, the primary function of the interface between the control device and the reader remains the same: that of commanding the reader to inventory tags and otherwise access tags for read, write, lock, and kill.

Readers that pass EPCglobal Gen 2 interoperability tests are designed to work seamlessly at the tag-to-reader interface. However, the interface at the network or controlling side of a reader has been radically different. Consequently, committing to a particular system might end up being more of a marriage to proprietary technology than originally intended. With proprietary control software in place, the cost and time associated with changing to alternative readers or controlling software could be prohibitive. In addition to cost and time concerns, proprietary technology defeats the plug and play strategy that is critical to technology innovation and wide-spread technology adoption. Because LLRP removes the task of writing numerous proprietary protocol interfaces that all support the same functionality, controlling application software vendors can focus on customer-visible features that impact RFID ROI.

2.2 Connection Details

LLRP is a binary protocol over TCP/IP, the Internet standard transport protocol. It is an asymmetric protocol where a client implementation of LLRP (hosted in the application software) sends commands to a Reader implementation of LLRP (hosted in the reader). The Reader

implementation answers commands by using responses, and also generates asynchronous data called events and reports.

LLRP supports reader- or client-initiated connections. By default Speedway will operate with client-initiated connections, waiting until the application has connected on the well-known TCP port 5084. In reader-initiated mode, the Reader will actively maintain connection to the host application. If the connection is broken for any reason, the Reader will continually try to reconnect, re-establishing connection without intervention from the host.

LLRP does not specify the behavior when the connection is broken. The Reader will continue to operate and collect tag data. The collected data will be (optionally) delivered upon the resumption of the connection if the *HoldEventsAndReports* field has been set in the LLRP configuration.

Readers only allow one LLRP connection at any time. Attempts to connect when an existing connection is active and healthy, results in a standard LLRP message sent back to the application, which reporting that another connection is currently active.

2.3 Declarative Specifications

One key advantage for LLRP is that it does not require real-time interaction between the application software and the Reader. The Reader can still be commanded to perform all of the Gen 2 time-critical functions. The Reader application software passes operational rules to the Reader in non-real time, and then triggers those rules to activate in real-time. The triggers can come from the application directly, from timers, General Purpose Input (GPI) hardware, or any other trigger defined by the Reader. This declarative operation method allows for the Reader to achieve peak performance without constraints caused by network or host latency. For applications, such as pharmaceutical lines where functions are time-critical, this more autonomous procedure facilitates uninterrupted line operation.

Alternatively, for applications such as a dock door in an isolated area where accommodating a controlling computer is not feasible, the ability to remotely control the Reader operation via common network connections provides cost savings and security advantages.

To enable this functionality, LLRP defines a set of Specifications (or Specs for short). The client sends these Specs to the Reader using the predefined messages and parameters discussed in Section 2.4. A spec contains all the information required to execute an operation on the Reader as well as special information about when the specs should activate and deactivate the operation. Rather than a single command that the Reader performs immediately, an LLRP specification (Spec) can be thought of as a document that describes the 'what, how and when' of operations the Reader performs.

LLRP also supports immediate triggers for these Specs which allow LLRP to operate in a more imperative mode, giving the application real-time control over the Reader operations.

2.4 Messages, Parameters, and Fields

The LLRP protocol can be described in terms of messages, parameters, and fields.

Messages are the unit of communications between the client and Reader. A message signifies a unique action requested by the client (such as ADD_ROSPEC) or an event from the Reader (RO_ACCESS_REPORT). Messages are defined as either client-initiated (typically called commands) or Reader-initiated (typically called responses or events). LLRP contains forty basic messages including all commands, responses, and events, plus a CUSTOM_MESSAGE to allow vendor extensions. Data accompanying a message is stored inside the message as either fields or parameters. Table 2.1 contains a list of all LLRP messages (except for responses), information about whether the message will solicit a response, which side of the LLRP connection initiates the message, and a short description of the purpose of the message.

2.4.1 Messages

Table 2.1 LLRP Messages

Message Name	Resp	Client/ Reader	Description
ADD_ACCESSSPEC	Y	C	Adds an AccessSpec to the reader
ADD_ROSPEC	Y	C	Adds an ROSpec to the Reader
CLIENT_REQUEST_OP	Y	R	Not supported by Impinj LTK
CLOSE_CONNECTION	Y	R	Reader generates this message before a client initiated connection closure
CUSTOM_MESSAGE	N/A	R or C	A custom message wrapper defined by LLRP to hold all custom message communication between the client and reader
DELETE_ACCESSSPEC	Y	C	Deletes an *AccessSpec* from the Reader
DELETE_ROSPEC	Y	C	Deletes an *ROSpec* from the Reader
DISABLE_ACCESSSPEC	Y	C	Disables an *AccessSpec* on the Reader
DISABLE_ROSPEC	Y	C	Disables an *ROSpec* on the Reader
ENABLE_ACCESSSPEC	Y	C	Enables an *AccessSpec* on the Reader
ENABLE_EVENTS_-AND_REPORTS	N	C	A message generated by the client to enable the RO_ACCESS_REPORT and READER_EVENT_NOTIFICATION. Only required by client when using the *HoldEventsAndReports* feature of LLRP.
ENABLE_ROSPEC	Y	C	Enables an *ROSpec* on the Reader

Message Name	Resp	Client/ Reader	Description
ERROR_MESSAGE	N	R	A message generated by the Reader when it is unable to properly decode and respond to a client message.
GET_ACCESSSPECS	Y	C	Gets the Reader's currently configured *AccessSpecs*
GET_READER_-CAPABILITIES	Y	C	Gets the Readers capabilities
GET_READER_-CONFIG	Y	C	Gets the Readers configuration
GET_REPORT	N	C	A message sent by the client to trigger a report to be generated from the Reader. This is in addition to any report triggers configured in the *ROSpec* or *AccessSpec*.
GET_ROSPECS	Y	C	Gets the Reader's currently configured *ROSpecs*
KEEP_ALIVE	Y	R	Reader periodically generates this message (when configured by the client). The response to this message generated by the client is called a KEEP_ALIVE_ACK (as opposed to the normal response nomenclature of LLRP)
READER_EVENT_-NOTIFICATION	N	R	A message generated by the reader to post asynchronous reader events (as opposed to tag events) to the client.
RO_ACCESS_REPORT	A	R	The report containing tag inventory and access data
SET_READER_CONFIG	Y	C	Sets the Reader configuration
START_ROSPEC	Y	C	Starts (activate) and *ROSpec* on the Reader
STOP_ROSPEC	Y	C	Stops (deactivates) an *ROSpec* on the Reader

2.4.2 Fields

Fields are individual data elements with a known format. LLRP has ten basic types listed in Table 2.2 below for reference. In addition to these basic types, LLRP defines fields that contain lists (or

vectors) of some these basic types, which are also noted in Table 2.2 below. When constructing LLRP messages/parameters, all fields must be present and within their valid range, although some fields are conditionally ignored.

Table 2.2 LLRP Basic Field Types

Type	Lists	Example(s)	Description
Bit/Boolean	**Y**	*CanDoRfSurvey*	Holds a Boolean value
TwoBits	**N**	*Session, MB*	Store a 2 bit unsigned integer
Bit Array	**Y**	*TagMask, TagData, EPC*	Hold a collection of data bits
Byte	**Y**	*ReaderID, ProtocolID (list)*	Holds an 8-bit value. Used most data arrays
Signed Byte	**Y**	*PeakRSSI AverageRSSI*	Holds a signed 8-bit value.
Signed Short Integer	**N**	*AntennaGain*	Holds a 16-bit signed integer
Unsigned Integer	**Y**	*ROSpecID, Frequency(list)*	Holds a 32-bit unsigned integer
Unsigned Short Integer	**Y**	*AntennaID, HopTableID, WriteData(list), ReadData(list)*	Holds a 16-bit unsigned integer
Unsigned Long Integer	**N**	*Microseconds*	Holds a 64-bit unsigned integer
UTF8 String	**N**	*FirmwareVersion*	Holds a printable (non-null-terminated) string

2.4.3 Parameters

Parameters are named data elements that can contain other parameters and/or fields. There are two distinctions between parameters and fields. Parameters can be optional elements that do not need to be present within the enclosing message or parameter. Parameters have a complex structure unlike the simple field types and can contain other parameters and fields. LLRP has numerous (nearly one hundred) defined parameters. Each parameter is uniquely identified by a parameter ID.

2.5 Introduction to LLRP Specs

LLRP defines Specs (Section 2.3) to control reader operation, antenna inventory cycles, tag access, and tag data reporting. This section introduces the two key specs supported in LLRP: *ROSpecs*

and *AccessSpecs.* There are other parameter specs within LLRP, but for the most part, they are enclosed within one of these main specs.

2.5.1 Reader Operation Specs (ROSpecs)

ROSpecs control the operation of the Reader. A *ROSpec* can be thought of as a description of the Inventory operations that the Reader is requested to perform. **The Reader supports one *ROSpec* at a time.** An *ROSpec* contains the following elements. Unless otherwise noted, all elements are mandatory.

- **ROSpecID** – an ID set by the client application to uniquely identify this spec in subsequent commands.
- **Priority** – must always be set to 0.
- **CurrentState** – The current state of the *ROSpec.* When the application is adding **ROSpecs**, this state must be set to **Disabled**.
- **ROBoundarySpec** – A description of the start and stop conditions for this operation.
- **ListofSpecs** – A list of one or more *AntennaInventorySpecs* to execute when this *ROSpec* activates.
- **ROReportSpec** (optional) – a parameter that describes when tag reports should be forwarded to the application and what data they should contain. This parameter is optional; if the value is not present, the reports generate according to the Reader's current settings available through the GET_READER_CONFIG message.

The *ROBoundarySpec* contains a start and stop trigger to define when the *ROSpec* should activate and deactivate on the Reader. The start trigger supports the following start conditions:

- **None/Null** – The enabled *ROSpec* will not start unless started by the client application via the START_ROSPEC message.
- **Immediate** – The *ROSpec* starts immediately once enabled. Note: with this trigger condition, an enabled *ROSpec* continuously restarts itself until disabled by the application using the DISABLE_ROSPEC message.
- **Periodic** – The enabled *ROSpec* is triggered to start periodically. This allows operation where an application directs the Reader to inventory tags for 10 seconds (such as cabinet or shelf) every 5 minutes and reports the results back at the end of each 10 second read internal.
- **GPI –** The enabled *ROSpec* triggers a start when the GPI event enters a certain state (such as **TRUE/HIGH**).

The stop trigger supports the following stop conditions:

- **None/Null** – The enabled *ROSpec* only stops when stopped by the client application via the STOP_ROSPEC message. Note: the *ROSpec* also stops when all of its enclosing operations (*AntennaInventorySpecs*) complete.
- **Duration** – The enabled *ROSpe***c** stops when it has been active for a specified duration.
- **GPI** – The enabled *ROSpec* triggers a stop when a GPI event enters a certain state (for example **TRUE/HIGH**).

> **Impinj Best Practice!** *Even if you are using complex start and stop triggers, an LLRP Reader always responds to the START_ROSPEC and STOP_ROSPEC messages. This is often very useful for manually triggering complex specs for testing within your application.*

The main RFID configuration for the inventory operation lies within the *AntennaInventorySpecs* (called *AISpec* for shorthand). Within this parameter, the application must specify the antenna set and the protocol (Gen2). Beyond that, all other RFID configuration (such as transmit power) can be set via the Speedway configuration. If desired, it can also be specified within the *AISpec* parameter.

Typically, a *ROSpec* contains a single *AISpec*. This spec is usually configured to enable all antenna ports (that have antennas connected) and uses the default Gen2 and RF configuration as specified in the SET_READER_CONFIG message. However, a *ROSpec* might contain multiple *AISpecs*. Multiple *AISpecs* can be programmed into the Reader for more complex behavior that is executed autonomously and without intervention from the application. An application can include multiple *AISpec***s** within a *ROSpec* for the following reasons:

- The application requires tight timing or sequence control over antennas. This could be executed by several *AISpecs,* each containing one or more antennas.

> **Impinj Best Practice!** *We recommend allowing the Speedway to automatically select its antenna, unless the particular use case dictates specific antenna switching times.*

- The application requires different RF configurations at different times. One example is a use case where the application first reads at low power and then reads at higher power to find all of its related case tags. A programming option would be to use two *AISpecs* within a single *ROSpec* that would execute automatically by the Speedway. This does not apply to the xArray Gateway.

The application requires different Gen2 parameters at different times. One example of this is a use case where, upon a trigger, the application first tries to read pallet tags by applying a Gen2

inventory filter. When no new pallet tags are visible, the application then instructs the Reader to search for all tags (or all case tags). One programming option for this scenario would be to use two *AISpecs* within a single *ROSpec* that would automatically be processed by the Reader.

When multiple *AISpecs* are contained within a *ROSpec*, the Speedway handles them as follows:

- The Speedway applies the configuration contained in the first *AISpec* as soon as the *ROSpec-StartTrigger* is satisfied, and begins the inventory operation.
- When that operation completes, defined by the stop trigger in the *AISpec*, the Reader applies the configuration contained in the second *AISpec* and begins that inventory operation.
- This process continues until one of the following two things occurs, at which the *ROSpec* stops automatically.

  All *AISpecs* complete.
  The *ROSpec* *ROSpecStopTrigger* is satisfied.

The client uses the LLRP messages and manages the *ROSpecs* :

- ADD_ROSPEC – Adds a *ROSpec* to the Reader.
- DELETE_ROSPEC – Deletes a *ROSpec* from the Reader.
- ENABLE_ROSPEC – Enables a *ROSpec* and activates when start conditions are met.
- DISABLE_ROSPEC – Disables a *ROSpec* and prevents activation even in the presence of start conditions.
- START_ROSPEC – Triggers the enabled *ROSpec* to start. This enables the client application with the ability to start the *ROSpec* manually.
- STOP_ROSPEC – Triggers the enabled *ROSpec* to stop. This enables the client application with the ability to stop the *ROSpec* manually.
- GET_ROSPEC – Gets a list of current *ROSpecs* provisions on the Reader.

Each of these client-initiated commands has a corresponding response sent by the Reader that reports the success or error status of the request (for example, the ADD_ROSPEC_RESPONSE) for a total of 14 messages that control *ROSpecs*.

2.5.2 Access Specs

LLRP *AccessSpecs* handles the extended tag operation of Gen2: read, write, lock, and kill. They provide an extensible mechanism which allows future Gen2 and proprietary tag operations.

During inventory, an LLRP Reader continually monitors the list of enabled *AccessSpecs*. When a spec matches certain conditions described below, the spec activates and the Reader performs the

operations contained within the spec. Based on the report configuration within the spec, results are generated and are sent to the client application.

AccessSpecs design supports multiple, different air protocols with unique commands. Because the Reader supports only Gen2 commands, this description simplifies the spec and conveys the general intent rather than the exact syntax.

A Gen2 *AccessSpec* contains:

- *AccessSpecID* – uniquely identifies this spec with an ID set by the client application.
- *AntennaID* – Specifies the set of antennas that this spec considers active. Specifying zero enables this spec for all antennas.
- *ProtocolID* – Specifies the protocol for access. This LLRP field allows support for other future protocols. Currently LLRP and Speedway support only Gen2.
- *CurrentState* – The current state of the *AccessSpec*. When the application adds *AccessSpec*, this state must be set to Disabled.
- *ROSpecID* – The *ROSpec* for which this spec should be considered active.
- *AccessSpecStopTrigger* – A stop condition that disables and deletes the AccessSpec itself. Used this for tag commissioning where the operation can only be performed a limited number of times (once).
- *AccessCommandOperation* – Performs the list of read, write, lock, and kill operations.
- *AccessReportSpec* – Defines how the access data reports to the application using this optional parameter.

> **Impinj Best Practice!:** *When building AccessSpecs, we recommend setting the AntennaID and ROSpecID to zero, which signifies all antennas/ROSpecs unless your use case has a specific reason to limit the spec to a single antenna or ROSpec. Carefully set the ROSpecID within the ROSpec to avoid mysteriously losing your application access functionality.*

AccessSpecs are managed by the client using the LLRP messages:

- ADD_ACCESSSPEC – Adds the *AccessSpec* to the Reader.
- DELETE_ACCESSSPEC – Deletes the *AccessSpec* from the Reader.
- ENABLE_ACCESSSPEC – Enables the *AccessSpec*. The Reader evaluates the enabled *AccessSpec* for execution during inventory.

- DISABLE_ACCESSSPEC – Disables the *AccessSpec*. When the *AccessSpec* is disabled, the Reader will not evaluate it for execution during inventory.
- GET_ACCESSSPEC – Gets the list of *AccessSpecs* provisions on the Reader.
- CLIENT_REQUEST_OP – Optional feature not supported.

Each of these client-initiated commands corresponds to a response sent by the Reader, which reports success or the error status of the request,for example ADD_ACCESSSPEC_RESPONSE for a total of 12 messages that control *AccessSpecs*.

For example, an application would add one or more *AccessSpecs* to:

- Write an EPC into a tag. In this instance, construct an *AccessSpec* that contains a write operation to write the EPC, a write operation to write a Gen2 password, and a lock operation to lock the EPC and password memory.
- Access user memory.
- Read the TID memory of the tag.
- Lock or kill a tag.

> **Impinj Best Practice!** *AccessSpecs have pre-defined priority. An LLRP-compliant Reader only executes the "first" matching AccessSpec for any given tag observation. The "first" AccessSpec is not defined by the AccessSpecID, but by the order in which the application adds the AccessSpecs to the Reader . When creating multiple AccessSpecs, we recommended choosing an AccessSpecID that reflects the order in which these were added. This information is not used by the Reader, but might be helpful for debugging and testing the application.*

2.6 A typical LLRP Exchange

A typical LLRP application involves these steps:

1. Application, or the Reader, establishes the connection.
2. Application requests the Reader's capabilities (optional). This allows the application to learn the unique characteristics of the Reader. Information about antennas (count), transmit power, frequency control, and LLRP features are available via the capabilities.

3. Application requests the Reader's configuration (optional). Some applications learn about the current Reader configuration before applying any new configuration values.

4. Application determines the configuration of the Reader, and changes the configuration if necessary (optional). This includes restoring the Reader to its LLRP default values, settings power, channel, and Gen2 default values.

    Impinj Best Practice! *Unless your application is specifically written to handle disconnected operation via LLRP, we recommend restoring the LLRP factory defaults when connecting to a Reader.*

5. Application adds and enables an ROSpec.

6. Application adds and enables multiple AccessSpecs (optional).

7. Application triggers *ROSpec* start and stop (optional).

8. Reader sends requested data report to the application.

A LTK example of this procedure is described in Section 6.5.3. A sample timeline of this exchange (steps 5-8) is shown in Figure 2.1.

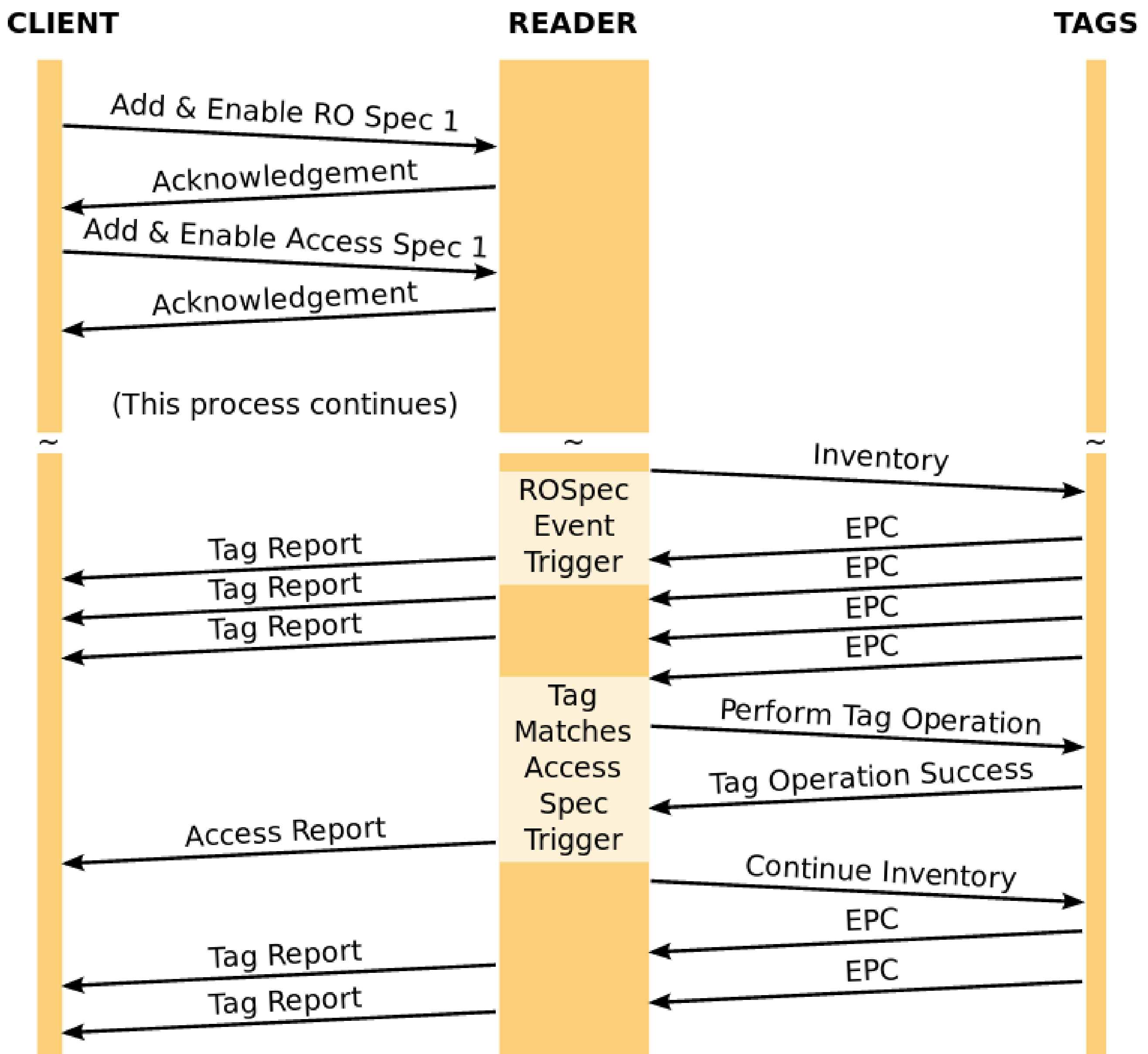


Figure 2.1 Sample LLRP Application Flow

2.7 Vendor Extensions to LLRP

Other standards have been proposed for the reader network interface, so why is LLRP successful? In part, it is because previous approaches did not go far enough to accommodate the needs of both reader and application software providers. These needs include the ability to better leverage the

competitive advantages of their respective products. In creating this new standard, the EPCglobal Reader Operations working group included a rich set of vendor extension points in LLRP. These extension points allow vendors the flexibility to innovate and differentiate their products, yet still operate within the standardized network framework. With clearly defined extension points, the controlling software can more easily expose unique extensions to particular products. End-users can also propose extensions for features specific to their enterprise.

2.8 Impinj LLRP Extensions

Current Reader functionality exceeds the standards proposed by the LLRP standards body. (For the standards description, see Section 1.6.1). Impinj Octane, the RFID firmware running on the Reader, offers differentiating features, which are not part of the standard LLRP specification via LLRP extensions and expand functionality.

Although it is possible to use the Reader without any Impinj Octane extensions to LLRP, learning to use the extensions available through Octane LLRP enables advanced features. These advanced features make configuration easier and extracts more performance from your RFID application.

Impinj extensions are described in the Octane LLRP reference guide, in Section 1.6.2.

3 LTK Overview

3.1 Origin and Purpose

Shortly after the ratification of the LLRP standard, a group of industry members met to see how they could help foster LLRP adoption. Both reader and application partners discussed what the key challenges were for LLRP adoption.

The group reached consensus that the LLRP protocol was necessarily more complex than any single company's proprietary reader protocol. It allows many different options to support many different reader variants. It also allows imperative or declarative operation, which require a complex syntax to communicate the client's intent.

The group also realized that these complexities could be overcome by building library support to construct and parse LLRP message. Several vendors contributed to the libraries, which supported the group's requirements to:

- Simplify LLRP message construction.
- Provide a validated machine description of LLRP.
- Support the common programming languages.
- Provide basic connection services.
- Enable a common framework for vendor extensions.
- Provide open source for use within a commercial product.
- Avoid the implementation of any application specific features, for fear they would become biased towards certain applications or frameworks.
- Provide a human-readable version of LLRP to facilitate an LLRP expert community.

The llrp-toolkit (LTK) is developed on source-forge (Section 1.6.6) and was released for general availability in November of 2008. The LTK represents the contributions of several industry leading companies and research labs.

3.2 LTK versus LLRP

Because the goal was to simplify message construction, not to abstract LLRP functionality, the LTK APIs translate in a 1:1 manner with LLRP message, parameters and fields. For example, a message ADD_ROSPEC has a corresponding class CADD_ROSPEC in the C++ LTK (LTKCPP) and a MSG_ADD_ROSPEC class in C#.NET (LTKNET). LLRP basic field types are translated to the most appropriate type for the individual languages. For example, in the C# LTK (LTKNET), the LLRP unsigned integer type is represented by a C# Uint32.

LTK offers some additional functionality that is not specified by the LLRP standard. Each LTK offers support for open and closing connections with the Reader, for performing message transactions that automatically wait for the Reader's response, and for reporting errors during message construction and parsing.

3.3 LTK Definition Files

LTK was required by its participants to provide both a machine description of LLRP as well as a framework for vendor extensions. LTK definition files were created for these purposes. For programmers using the pre-built Impinj LTK, these definition files have been integrated into the release packages. Subsections 3.3 and 3.4 are not necessary reading and can be skimmed.

> **Impinj Best Practice!** *Unless you are using languages not supported by Impinj, we recommend using our pre-built Impinj LTK distributions.*

A single machine description file **'llrp-1x0-def.xml'** describes the messages and parameters in the LLRP standard protocol (version 1.0.1). Each toolkit library parses this file and builds the internal LLRP object representations. LTKC, LTKCPP, LTKJAVA, and LTKNET process this definition file at compile time to build the classes and methods to support building and parsing the LLRP message. In addition to containing the machine description of the protocol, the file also contains embedded xhtml documentation and links to the LLRP standard for use by auto-generated documentation and tooltips.

An example excerpt from the LLRP specification and the LTK **'llrp-1x0-def.xml'** file for the *UTCTimestampParameter* is shown below:

UTCTimestamp **Parameter**

- Compliance requirement: Compliant Readers and Clients that have UTC clocks SHALL implement this parameter.

UTCTimestamp 7.1.3.1.1.1

- *MicroSeconds: Unsigned Long Integer.* This is the field time elapsed since the Epoch (00:00:00 UTC, January 1, 1970) measured in microseconds.

2868 16.2.2.1

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						Type=128										Parameter Length															
Microseconds [63:32]																															
Microseconds [31:01]																															

Copyright 2006, 2007 EPCglobal, Inc

Figure 2.2 UTCTimestamp Parameter

These two excerpts above precisely describe the contents and format of this parameter. However, there's no easy way to translate this text/diagram into software that can build and parse messages. Below is the same parameter description in the LTK description file format.

LLRP-DEF Machine Description

```
<parameterDefinition name="UTCTimestamp" typeNum="128" required="false">
    <annotation>
      <documentation>
        <h:a href="http://www.epcglobalinc.org/standards/llrp/
            llrp_1_0_1-standard-20070813.pdf#page=37&amp;view=fit">
            LLRP Specification Section 7.1.3.1.1.1</h:a>
        <h:a href="http://www.epcglobalinc.org/standards/llrp/
        llrp_1_0_1-standard-20070813.pdf#page=131&amp;view=fit">
            LLRP Specification Section 16.2.2.1</h:a>
      </documentation>
      <description copyright="Copyright 2006, 2007, EPCglobal Inc. The
          proprietary text of EPCglobal Inc. included herein is not a
          Contribution to the LLRP toolkit, under Apache License,
          Version 2.0. The right to use the proprietary text is limited to
          reproduction and display thereof within the work.">
        <h:p> The timestamps in LLRP messages or parameters can be either
          the uptime or the UTC time [UTC]. If a Reader has a UTC clock, all
          timestamps reported by the Reader <h:b>SHALL</h:b> use an UTC
          timestamp parameter. If a Reader has no UTC clock capability, all
          timestamps reported by the Reader. <h:b>SHALL</h:b> use the
          uptime parameter.</h:p>
      </description>
    </annotation>
```

```
    <field type="u64" name="Microseconds" format="Datetime"/>
  </parameterDefinition>
```

Figure 3.1 LTK Parameter Machine Description

3.4 LTK Vendor Extensions

To support vendor extensions, a schema **'llrp-1x0.xsd'** describes the language used to specify the **'llrp-1x0-def.xml'** machine description of the LLRP protocol. This same language can be used to specify custom extension to LLRP. Vendors that write their own definition files conformant to the **'llrp-1x0.xsd'** schema can get support for their LLRP extension through LTK. An example excerpt from the Impinj vendor extension file **'ImpinjDef.xml'** describes the IMPINJ_ENABLE_EXTENSIONS_RESPONSE message is shown below.

LLRP-DEF

```
<customMessageDefinition    name="IMPINJ_ENABLE_EXTENSIONS"
                            vendor="Impinj" subtype="21"
                            namespace="Impinj"
                            responseType="IMPINJ_ENABLE_EXTENSIONS_RESPONSE">
    <annotation>
      <documentation>
      </documentation>
      <description copyright="Copyright 2007, 2008 Impinj Inc.">
        <h:p>The IMPINJ_ENABLE_EXTENSIONS message only applies for the duration
        of the current LLRP connection. If the LLRP connection is broken and
        re-established, the application must re-issue this command.</h:p>
      </description>
    </annotation>
    <reserved   bitCount="32"/>
    <parameter  repeat="0-N"    type="Custom"/>
</customMessageDefinition>
```

Figure 3.2 LTK Vendor Extension Machine Description

3.5 LTK-XML

It is often useful to share the details of your LLRP specs with another party, such as when you contact technical support. However, LLRP is a binary protocol, which makes the sharing of

messages in a human-readable form difficult. The LTK allows conversion of any LLRP message or parameter into LTK-XML for easy viewing and sharing via e-mail. LTK-XML is an XML-based representation of the LLRP message contents. There is an exact one-to-one mapping between any LTK-XML message and its corresponding binary LLRP message.

For example, a SET_READER_CONFIG message that enables GPI (3) can be displayed in the native LLRP binary format, compactly and efficiently as:

LLRP-HEXBINARY

```
04 03 00 00 00 13 00 00 00 00 00 00 E1 00 08 00 03 80 02
```

Figure 3.3 LLRP Binary SET_READER_CONFIG Message

However, this is difficult to read and troublesome to interpret or change. To make it human-readable, you could add some annotations such as the following:

LLRP-HEXBINARY (annotated)

Hdr	Msg	Length	MessageID	Fac	Type	Len	GPI #	Cfg	State
04	03	00 00 00 13	00 00 00 00	00	00 E1	00 08	00 03	80	02

Figure 3.4 Annotated SET_READER_CONFIG Message

This annotated message describes the *MessageType* (0x03), the **GPIPortCurrentState** Parameter (0xe1), and the other fields of the message. This is still difficult to interpret, and removes the machine readability of the underlying binary, making it hard to validate or utilize.

We can express the same SET_READER_CONFIG message in the LTK-XML as:

LTK-XML

```xml
<SET_READER_CONFIG Version="1" MessageID="0">
    <ResetToFactoryDefault>False</ResetToFactoryDefault>
    <GPIPortCurrentState>
        <GPIPortNum>3</GPIPortNum>
        <Config>True</Config>
        <State>Unknown</State>
    </GPIPortCurrentState>
</SET_READER_CONFIG>
```

Figure 3.5 LTK-XML SET_READER_CONFIG Message

All LTK libraries can automatically generate and parse LTK-XML representations from their internal representations of LLRP messages and parameters. Some LTK programming languages like LTKPerl use LTK-XML exclusively to build and parse LLRP messages. The pre-built Impinj LTKNET and Impinj LTKCPP (Linux only) allows either object-based or XML-based message generation.

Throughout this document, examples of LLRP messages and parameters will use the LTK-XML version of the parameters because they are readily human-readable and are directly usable by some of the LTK libraries. Binary LLRP will not be shown.

Note: LLRP was chosen as a binary protocol for several reasons. Binary protocols are efficient on the wire interface, and are more easily implemented by smaller processors. All communications between the LTK and the Speedway occurs in binary form.

> **Best Practice**: *When you submit support requests while you are programming with the LTK, we recommended that you include the LTK-XML copies of your ROSpecs and AccessSpecs. This gives our support team the information they need to analyze your issue.*

4 Getting Started with the Impinj LTK

This section gives a quick overview for starting and using the Impinj LTK.

4.1 Overview

4.1.1 Impinj Supported Platforms and Languages

LTKC library – This library is suitable for use in Linux-based ANSI C programs that require LLRP support. Impinj distributes periodic releases of this toolkit. The releases are named **'libltkc__sdk__X__Y__Z.tgz'.** Each release includes static Linux libraries for x86 and Arm platforms. The ARM library is compatible with Speedway Readers and xArray Gateways. The x86 library is compatible with most 2.6 Linux systems and is useful for debugging and testing embedded applications via a host processor. This library also includes libxml2 libraries to support LTK-XML decoding for Linux platforms only.

LTKCPP Library – This library is suitable for use in Linux- or Windows-based C++ applications that require LLRP support. The release includes static Linux and Windows libraries. Linux libraries are released for x86 and Arm platforms. The Linux Arm library is compatible with Speedway Readers and xArray Gateways. The Linux x86 library is compatible with most 2.6 Linux systems and is useful for debugging and testing embedded applications via a host processor. Windows libraries are compatible with x86 only. This library also includes libxml2 libraries to support LTK-XML decoding for Linux platforms only. Two version of this archive file are released, a **'libltkcpp__sdk__X__Y__Z.zip'** most suited to Windows platforms, and a **'libltkcpp__sdk__X__Y__Z.tgz'** version most suited to Linux platforms. Other than line endings in the header files, these two archives are identical.

LTKNET Library – This library is suitable for use in the Windows C# .NET development environment. This release includes dynamic managed assemblies (.dlls) for the C# .NET framework. A single version of this zip file, **'libltknet__sdk__X__Y__X.zip'** is available.

LTKJava Library – This library is suitable for use with Java virtual machines (version 1.5 and later). The release includes jar files with and without Java dependencies. Two versions of this archive are released, a **'libltkjava__X.Y.Z.zip'** most suited to Windows platforms, and a **'libltkjava__X.Y.Z.tgz'** version most suited to Linux platforms. Other than line endings in the header files, these two archives are identical.

> **Impinj Best Practice**! *Due to better support and documentation, we recommend using the LTKCPP library instead of LTKC, unless C support is required.*

4.1.2 CHANGES.TXT

All Impinj LTK releases are accompanied by a **CHANGES.TXT** file which chronicles recent changes to that particular LTK release with notes on new features, bug fixes, and more. The file also contains the known defect list at the time of the release.

> **Impinj Best Practice**! *We strongly recommended reading the* **CHANGES.TXT** *file before starting integration with an existing application.*

4.1.3 Version Names

In the release name examples above, such as *'libltknet_sdk_X_Y_X.zip'**, the version numbers are represented as follows:

- "X" denotes the major release version. Changes in the major version represent significant updates that are not necessarily backward compatible with previous updates.
- "Y" denotes the minor release version. Changes in the minor version represent API changes and additions that have backward compatibility with previous versions.
- "Z" denotes the maintenance release version. Changes in the maintenance version represent fully backward compatible bug fixes and do not affect the programming APIs.

We recommend a re-compile and re-link of your application for all new updates of the LTK.

4.2 Obtaining the LTK

The LTK libraries are available from `http://support.impinj.com`, and are available in standard .zip and .tgz archive formats.

4.3 LTK Required Tools/Environment

Compiler tools are required in order to compile your application and link to the SDK. These tools include:

LTKC –x86 Linux requires gcc. On-reader development requires the xScale embedded SDK based on gcc or based on the Impinj ARM.

LTKCPP –x86 Linux requires gcc. x86 Windows requires Microsoft Visual Studio 2005 C++ or later. On-reader development requires the Impinj ARM or the xScale embedded SDK based on gcc.

LTKNET –LTKNET requires Microsoft Visual Studio 2005 C# or later.

LTKJAVA – Java JDK version 1.5 or later is required. Ant version 1.8 or later is recommended.

ALL–The Reader requires the LTK sample applications to run. Ensure that your Reader connects to the network and is accessible by hostname from your target application machine.

For details about the Impinj ARM and xScale SDK tools, see Section 1.6.3.

4.4 Installing and Running the LTK Samples

Extract the archive into your code project. The archive contains all files in in a subdirectory named libltkxxx (libltkc, libltkcpp, libltkjava, and libltknet), where xxx is different for each library. Some libraries contain large directories of html-based documentation in the libltkxxx folder. Program compilation does not require these libraries.

Each library contains sample code that you can use to build and test the libraries on your system. We strongly recommend that you validate the library functions by running compile and the sample code on your system. The Instructions below show how to build and run this sample code.

4.4.1 LTKC

The LTKC comes with a single example that exercises the library with a basic LLRP connection and inventory. The steps to build and run the example are shown below. The **'makefile'** for this example attempts to compile the program for three platforms: x86-Linux, xScale-Linux, and Atmel-ARM9-Linux. If your host platform does not contain all necessary compilers, the compilation will fail.

LTKC

```
$ cd libltkc

$ cd example

$ make

cc -g -I.. -o example1 example1.c ../libltkc_x86.a

$ ls

example1 example1.c Makefile

$ ./example1 speedway-00-06-4b
```

```
INFO: Starting run 1 ================

NOTICE: Antenna 2 is disconnected

NOTICE: Antenna 3 is disconnected

NOTICE: Antenna 4 is disconnected

INFO: 2 tag report entries

3000-2141-60C0-0400-0000-6AB2

3008-33B2-DDD9-0580-3505-0000

[...]

INFO: Starting run 5 ================

NOTICE: Antenna 2 is disconnected

NOTICE: Antenna 3 is disconnected

NOTICE: Antenna 4 is disconnected

INFO: 1 tag report entries

3008-33B2-DDD9-0580-3505-0000

INFO: Done

$
```

Figure 4.1 LTKC – Sample Compilation and Execution

To build for a single platform only, use a single make target, **all_x86**, **all_xscale**, or **all_atmel**.

Note: To run the sample code requires a Reader networked to your development PC.

4.4.2 LTKCPP

The LTKCPP distributions beyond 10.6.x contain the sample code used in this documentation.

Unzipping (or un-tarring) the distribution creates two new directories: **'Documentation'** and **'libltkcpp'**. **'Documentation'** contains the HTML-based documentation for programming the LTK. Access the API documentation by opening the file **'\Documentation\html\index.html'** in your web browser.

The **'libltkcpp'** directory contains the library header files, library binary files, as well as the sample code referenced in this document. The documentation sample code builds using Linux makefile or Microsoft Visual Studio Solution. The main **'makefile'** for the documentation samples is **'libltkcpp/Makefile'**. The Visual Studio solution file for documentation samples is **'ImpinjExmples.sln'**.

Ensure that you have the proper tools installed (Section 4.3) to follow and build the Linux examples below. The examples compile for x86_linux, or arm-Linux. Use the make target **all_x86** to build for x86-Linux**, useall_atmel** to compile for the Speedway embedded arm platform, and everything to compile for all targets. Executable applications build in the docSampleX subdirectories and use suffixes to describe their target platform.

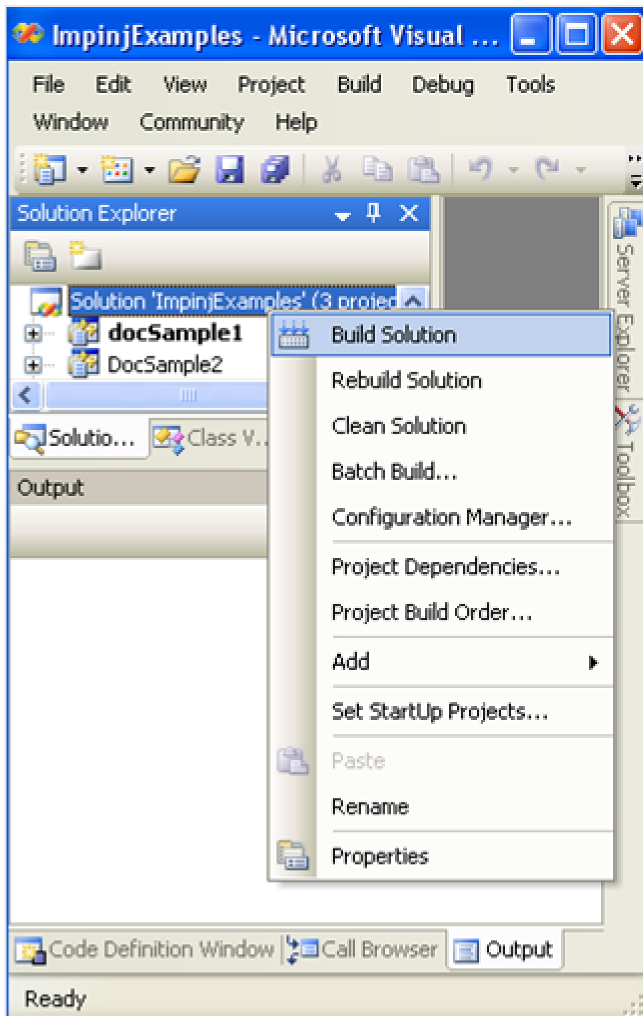
LTKCPP Linux

```
$ make all_x86

cd docsample1; make all_x86

make[1]: Entering directory `/home/impinj/Development/Temp/libltkcpp/docsample1`

make all AR=ar CXX=g++ CPPFLAGS="-g -Wall -I.." SUFFIX=_x86

...

make[1]: Leaving directory `/home/impinj/Development/Temp/libltkcpp/docsample3`

$
```

Figure 4.2 LTKCPP – Linux Makefile Sample Compilation and Execution

To build within Visual Studio, browse for the file **'ImpinjExmples.sln'** and open it with Visual Studio 2005 or later. Build the solution by right-clicking the solution, or using the **Build** tab at the top of the IDE.

LTKCPP Visual Studio


ImpinjExamples - Microsoft Visual ...
File
Edit
View
Project
Build
Debug
Tools
Window
Community
Help
Solution Explorer
Solution 'ImpinjExamples' (3 projec
docSample1
DocSample2
Solutio...
Class V...
Output
Build Solution
Rebuild Solution
Clean Solution
Batch Build...
Configuration Manager...
Project Dependencies...
Project Build Order...
Add
Set StartUp Projects...
Paste
Rename
Properties
Server Explorer
Toolbox
Code Definition Window
Call Browser
Output
Ready

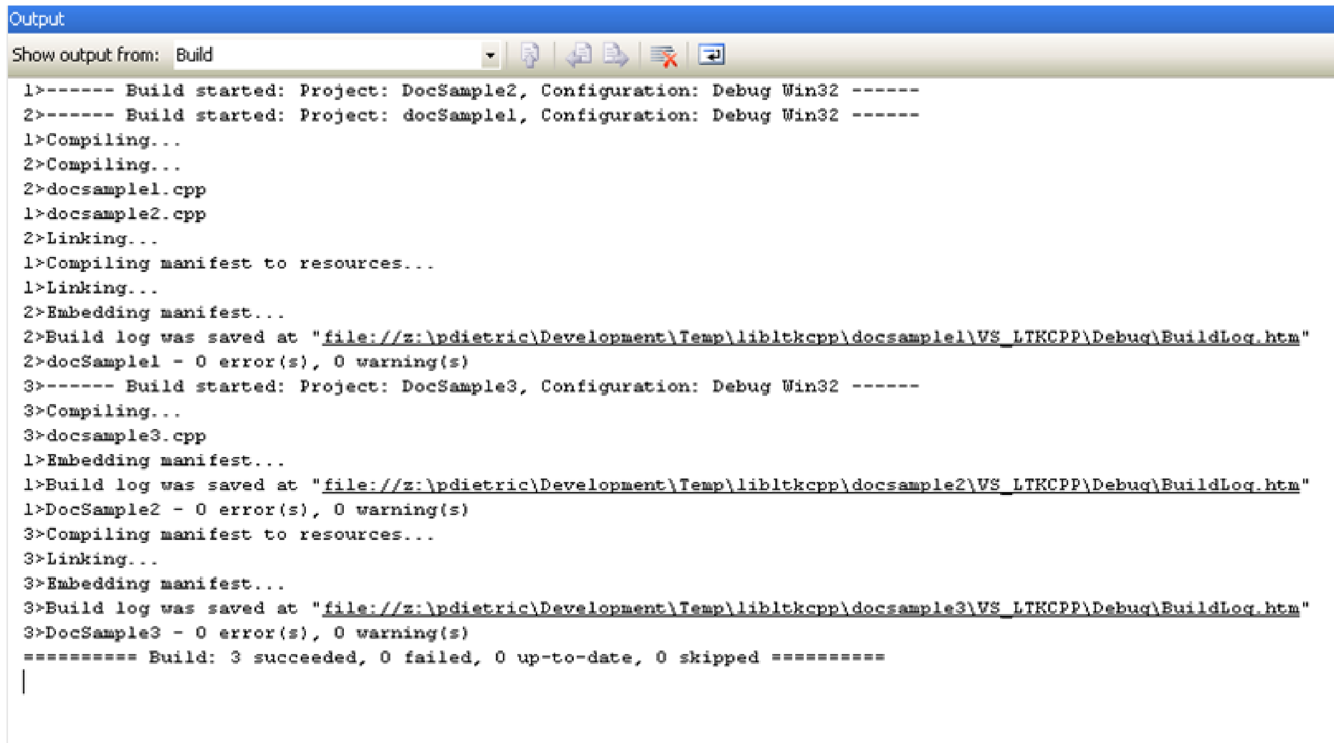


Figure 4.3 LTKCPP – Visual Studio Sample Compilation and Execution

You can run the application from the command line in either DOS or Linux.

LTKCPP Docsample1-x86

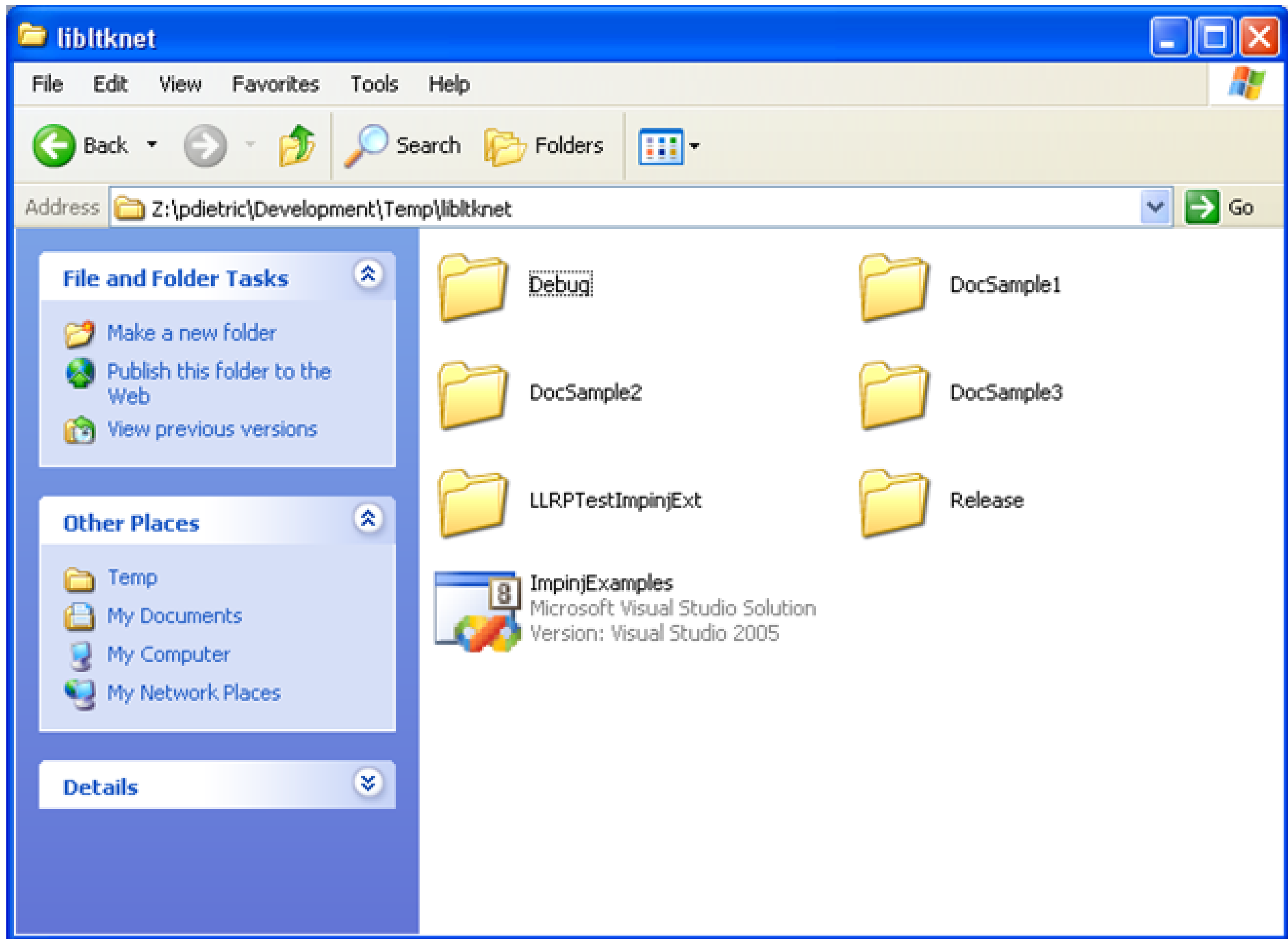
```
$ cd docsample1/

$ ./docsample1_x86 speedway-00-11-b6

EPC: AAAA-AAAA

EPC: BBBB-BBBB

EPC: EEEE-EEEE

EPC: BBBB-BBBB

EPC: DDDD-DDDD

EPC: EEEE-EEEE
```

```
EPC: AAAA-AAAA

EPC: BBBB-BBBB

EPC: EEEE-EEEE

NOTICE: Antenna 3 is disconnected

NOTICE: Antenna 4 is disconnected

EPC: DDDD-DDDD

EPC: EEEE-EEEE

...
```

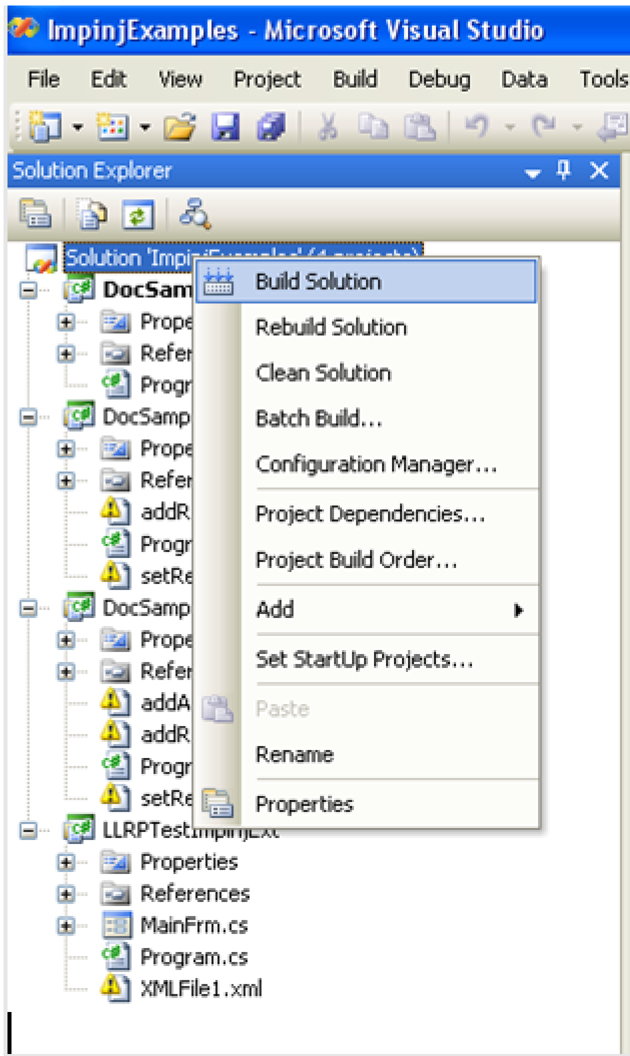
4.4.3 LTKNET

Unzip the archive file and browse to the **'libltknet'** directory. Open the **'ImpinjExamples.sln'** solution file to display an LTKNET example.

LTKNET



Build the samples by right-clicking the solution file, or by clicking Build at the top of the IDE page, as shown in Figure 4.4.


ImpinjExamples - Microsoft Visual Studio
File
Edit
View
Project
Build
Debug
Data
Tools
Solution Explorer
Build Solution
Rebuild Solution
Clean Solution
Batch Build...
Configuration Manager...
Project Dependencies...
Project Build Order...
Add
Set StartUp Projects...
Paste
Rename
Properties
Properties
References
MainFrm.cs
Program.cs
XMLFile1.xml

```
Output
Show output from: Build
------ Build started: Project: DocSample3, Configuration: Debug Any CPU ------
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 /d

Compile complete -- 0 errors, 0 warnings
DocSample3 -> Z:\pdietric\Development\Temp\libltknet\DocSample3\bin\Debug\DocSample3.exe
------ Build started: Project: DocSample2, Configuration: Debug Any CPU ------
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 /d

Compile complete -- 0 errors, 0 warnings
DocSample2 -> Z:\pdietric\Development\Temp\libltknet\DocSample2\bin\Debug\DocSample2.exe
------ Build started: Project: DocSample1, Configuration: Debug Any CPU ------
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 /d

Compile complete -- 0 errors, 0 warnings
DocSample1 -> Z:\pdietric\Development\Temp\libltknet\DocSample1\bin\Debug\DocSample1.exe
------ Build started: Project: LLRPTestImpinjExt, Configuration: Debug Any CPU ------
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 /d

Compile complete -- 0 errors, 0 warnings
LLRPTestImpinjExt -> Z:\pdietric\Development\Temp\libltknet\LLRPTestImpinjExt\bin\Debug\LLRPTest.exe
========== Build: 4 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Figure 4.4 LTKNET – Sample Compilation

Executables are placed in the **'Debug'** directories (for example **'DocSample1\bin\Debug'**) in the sample project. Run the samples by opening a DOS command window and executing the sample as shown in Figure 4.5.

LTKNET

```
C:\libltknet\DocSample1\bin\Debug>DocSample1.exe speedway-00-11-b6

Impinj C# LTK.NET RFID Application DocSample1 reader - speedway-00-11-b6

Initializing

Adding Event Handlers

Connecting To Reader

Enabling Impinj Extensions

Factory Default the Reader

Adding RoSpec

Enabling RoSpec
```

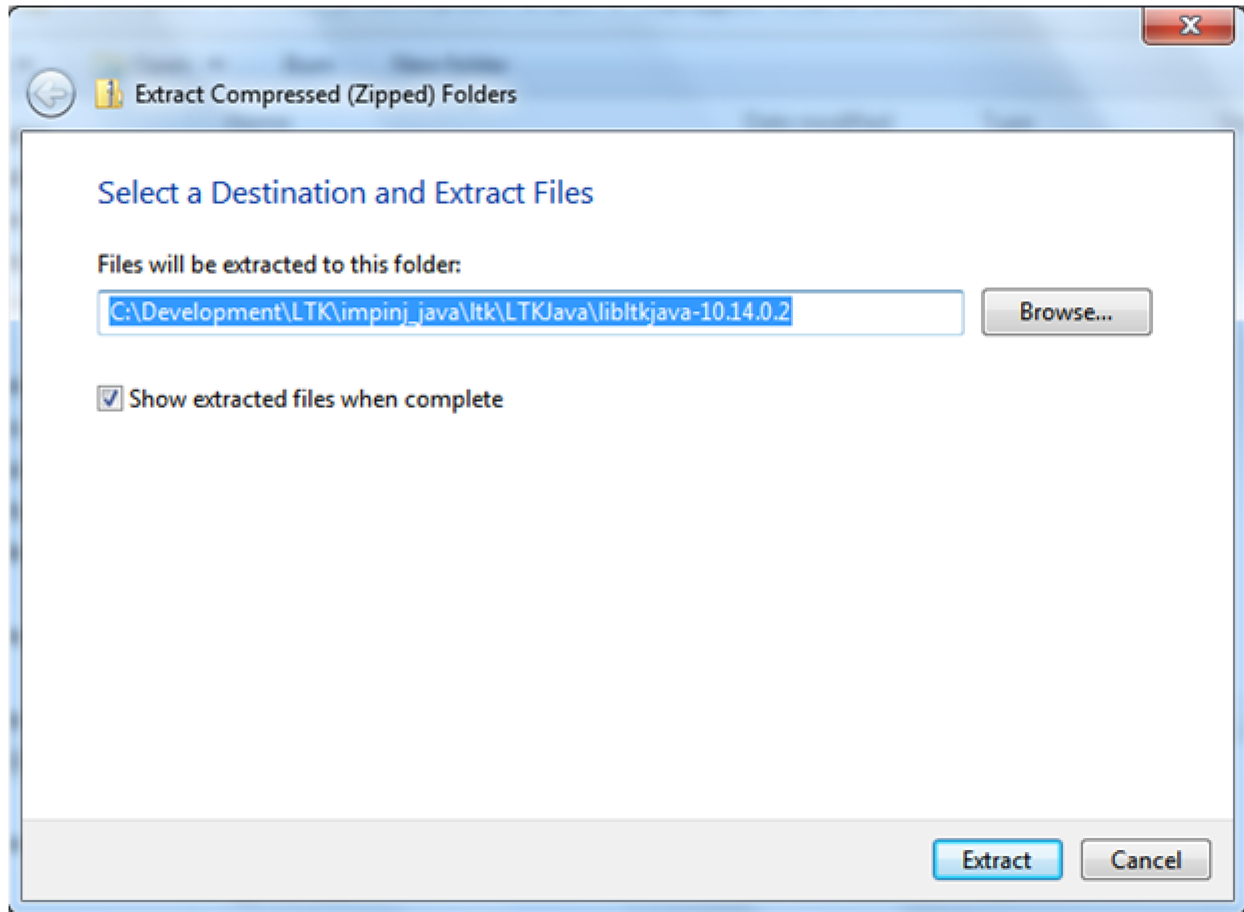
```
Starting RoSpec

EEEEEEEE

DDDDDDDD

BBBBBBBB

AAAAAAAA

======Reader Event 2======2009-05-22T22:28:02.9720000Z

======Reader Event 1======2009-05-22T22:28:02.9670000Z

<AntennaEvent>

   <EventType>Antenna_Disconnected</EventType>

   <AntennaID>3</AntennaID>

</AntennaEvent>

<AntennaEvent>

   <EventType>Antenna_Disconnected</EventType>

   <AntennaID>4</AntennaID>

</AntennaEvent>

EEEEEEEE

DDDDDDDD

AAAAAAAA

EEEEEEEE

. . .
```

Figure 4.5 LTKNET – Sample Execution

4.4.4 LTKJava

Windows

Download the zip file from http://support.impinj.com, and then extract the files from the archive, as shown in Figure 4.6.



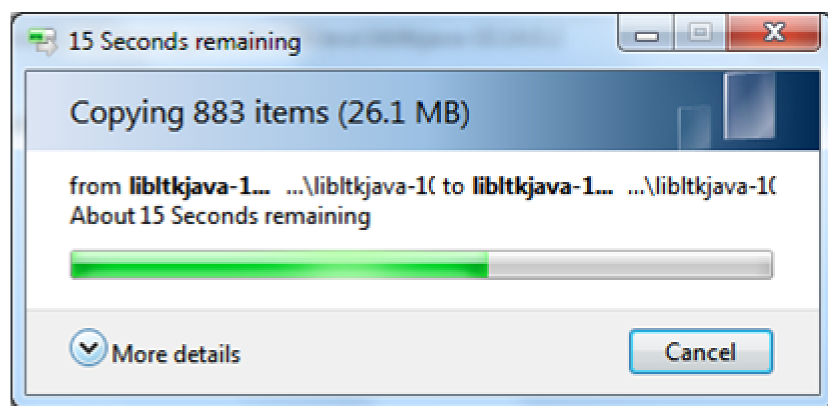
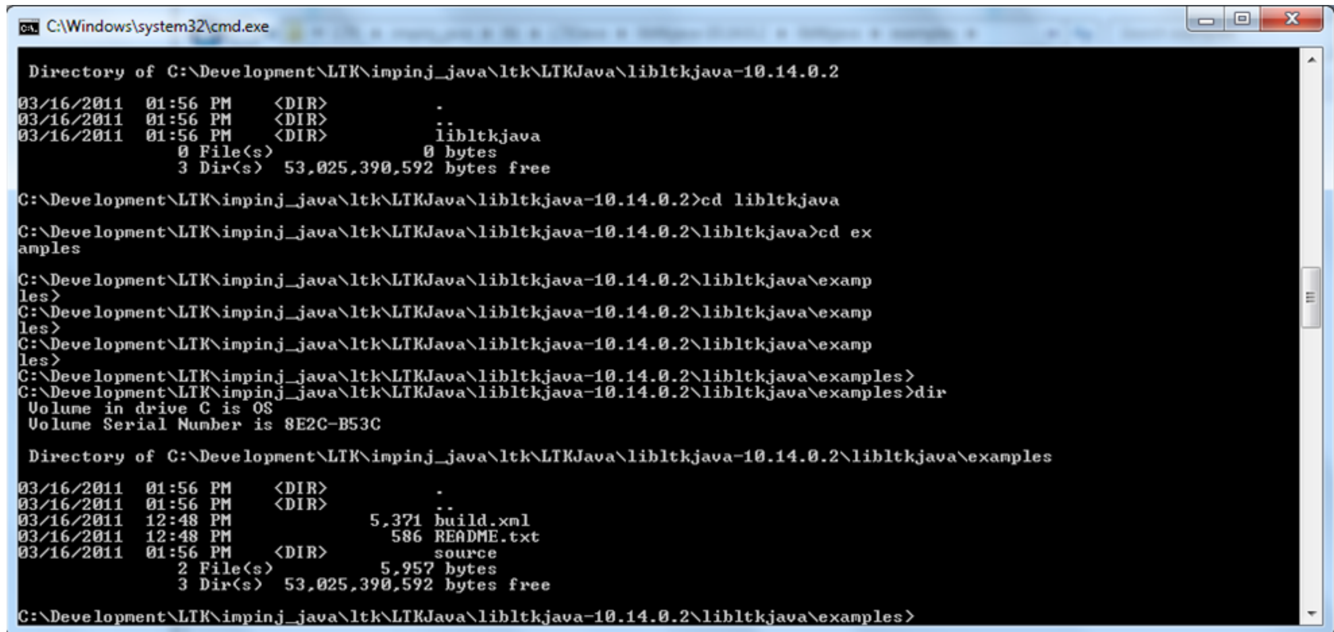


Figure 4.6 LTKJAVA – Windows Library Extraction

Open a DOS command window and 'cd' to the examples directory in the extracted directory tree.



Set up your path to ensure access to both JAVA and ant. Your path might look something like the example below. You might also need to set your JAVA_HOME environment variables as follows:

LTKJAVA – Windows

```
PATH=%PATH%;"c:\Program Files\Java\jdk1.6.0_21\bin";"c:\Program Files
\NetBeans 6.9.1\java\ant\bin"

set JAVA_HOME="C:\Progra~1\Java\jdk1.6.0_21"
```

Build the examples using the following command line:

LTKJAVA – Windows

```
ant all
```

The samples should compile and result in the following output below:

LTKJAVA – Windows

```
C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\libltkjava\
examples>ant

Buildfile

   C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\libltkjava\
   examples\build.xml

init:

   [echo] using ant version 'Apache Ant version 1.8.1 compiled on April 30 2010'

   [echo] using java version '1.6.0_21'

docsample1:

   [echo] Building DocSample1 ...

   [javac] Compiling 1 source file to C:\Development\LTK\impinj_java\ltk\
   LTKJava\libltkjava-10.14.0.2\libltkjava\examples\target\classes

docsample2:

   [echo] Building DocSample2 ...

   [javac] Compiling 1 source file to C:\Development\LTK\impinj_java\ltk
   \LTKJava\libltkjava-10.14.0.2\libltkjava\examples\target\classes

docsample3:

   [echo] Building DocSample3 ...

   [javac] Compiling 1 source file to C:\Development\LTK\impinj_java\ltk
   \LTKJava\libltkjava-10.14.0.2\libltkjava\examples\target\classes

docsample4:

   [echo] Building DocSample4 ...

   [javac] Compiling 1 source file to C:\Development\LTK\impinj_java\ltk\
```

```
   LTKJava\libltkjava-10.14.0.2\libltkjava\examples\target\classes

all:

BUILD SUCCESSFUL

Total time: 4 seconds

C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\
libltkjava\examples>
```

Figure 4.7 LTKJAVA – Windows Sample Compilation

Run the sample with the following command.

LTKJAVA – Windows

```
ant -Dreadername=192.168.1.5 run-docsample1
```

The output should appear similar to what is shown below. Replace 192.168.1.5 with the IP address or hostname of your Reader. Program output looks similar to the display below.

LTKJAVA - Windows

```
C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\
libltkjava\examples>ant -Dreadername=192.168.1.5 run-docsample1

Buildfile:

   C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\
   libltkjava\examples\build.xml

init:

   [echo] using ant version 'Apache Ant version 1.8.1 compiled on April 30 2010'

   [echo] using java version '1.6.0_21'

docsample1:

   [echo] Building DocSample1 ...
```

```
   [javac] Compiling 1 source file to C:\Development\LTK\impinj_java\ltk\
   LTKJava\libltkjava-10.14.0.2\libltkjava\examples\target\classes

run-docsample1:

   [echo] Running DocSample1 ...

   [java] 0 [main] INFO org.impinj.llrp.ltk.examples.docsample1 - Initiate
   LLRP connection to reader

   [java] 236 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   IMPINJ_ENABLE_EXTENSIONS ...

   [java] 262 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   IMPINJ_ENABLE_EXTENSIONS was successful

   [java] 262 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   SET_READER_CONFIG with factory default ...

   [java] 293 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   SET_READER_CONFIG Factory Default was successful

   [java] 293 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   Building ADD_ROSPEC message from scratch ...

   [java] 329 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   Sending ADD_ROSPEC message ...

   [java] 335 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   ADD_ROSPEC was successful

   [java] 350 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   ENABLE_ROSPEC ...

   [java] 354 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   ENABLE_ROSPEC was successful

   [java] 354 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   START_ROSPEC ...
```

```
   [java] 369 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   START_ROSPEC was successful

   [java] EPC: 330833b2ddd906c000000000 Antenna: 1 FirstSeen:
   2011-03-16T16:40:57.921457-04:00 RSSI: -34

   [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
   2011-03-16T16:40:57.931715-04:00 RSSI: -36

   [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
   2011-03-16T16:40:58.992678-04:00 RSSI: -39

   [java] EPC: 330833b2ddd906c000000000 Antenna: 1 FirstSeen:
   2011-03-16T16:40:59.190145-04:00 RSSI: -39

   [java] 2369 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   STOP_ROSPEC ...

   [java] 2383 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   STOP_ROSPEC was successful

   [java] 2391 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
   CLOSE_CONNECTION was successful

BUILD SUCCESSFUL

Total time: 4 seconds

C:\Development\LTK\impinj_java\ltk\LTKJava\libltkjava-10.14.0.2\
libltkjava\examples>
```

Figure 4.8 LTKJAVA – Windows Sample Execution

Linux

Download the compressed tar file from http://support.impinj.com. Extract files from the archive using the following commands:

LTKJAVA

```
tar -zxvf libltkjava-10.14.0.2.tgz
```

Figure 4.9 LTKJAVA – Linux Library Extraction

Note: The version number of the file you download reflects the current version of LTKJAVA.

Change to the **'libltkjava'** directory. Review the **'CHANGES.TXT'** file to view the latest known issues and revisions to LTKJAVA. Change to the examples directory.

LTKJAVA

```
cd libltkjava/examples
```

Set up your ant path and your ANT_HOME and JAVA_HOME environment variables.

Note: Your values will likely be different than the example, but these variables act as guides for setting up the correct environment.

LTKJAVA - Linux

```
export JAVA_HOME=/opt/ltk/jdk1.6.0_18/

export ANT_HOME=/opt/ltk/apache-ant-1.8.0RC1/

export PATH=$PATH:/opt/ltk/jdk1.6.0_18/bin:/opt/ltk/apache-ant-1.8.0RC1/bin/
```

Use the following code for building the examples using ant.

LTKJAVA - Linux

```
ant all
```

You should see output similar to that shown below. Individual programs can build using

```
'ant docsampleX'
```

where X is the sample number.

LTKJAVA - Linux

```
[revolution@RevolutionEDK examples]$ ant all

Buildfile:

   ./libltkjava/examples/build.xml

init:

   [echo] using ant version 'Apache Ant version 1.8.0RC1 compiled on
   January 5 2010'

   [echo] using java version '1.6.0_18'

docsample1:

   [echo] Building DocSample1 ...

   [javac] Compiling 1 source file to ./libltkjava/examples/target/classes

docsample2:

   [echo] Building DocSample2 ...

   [javac] Compiling 1 source file to ./libltkjava/examples/target/classes

docsample3:

   [echo] Building DocSample3 ...

   [javac] Compiling 1 source file to ./libltkjava/examples/target/classes

docsample4:

   [echo] Building DocSample4 ...

   [javac] Compiling 1 source file to ./libltkjava/examples/target/classes

all:

BUILD SUCCESSFUL
```

```
Total time: 2 seconds
```

Figure 4.10 LTKJava – Linux Sample Compilation

Running an example requires a Reader connected via Ethernet directly or indirectly to the computer running these examples. To run **docsample1**, enter:

LTKJAVA – Linux

```
ant -Dreadername="192.168.1.5" run-docsample1
```

Replace 192.168.1.5 with the IP address or hostname of your Reader. The program output looks similar to the display below.

LTKJAVA - Linux

```
[revolution@RevolutionEDK examples]$ ant -Dreadername="192.168.1.5"
run-docsample1

Buildfile:

   /mnt/hgfs/Development/LTK/impinj_java/ltk/LTKJava/libltkjava/
   examples/build.xml

init:

   [echo] using ant version 'Apache Ant version 1.8.0RC1 compiled on
   January 5 2010'

   [echo] using java version '1.6.0_18'

docsample1:

   [echo] Building DocSample1 ...

   [javac] Compiling 1 source file to

      /mnt/hgfs/Development/LTK/impinj_java/ltk/LTKJava/libltkjava/
      examples/target/classes

run-docsample1:
```

```
[echo] Running DocSample1 ...

[java] 0 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
Initiate LLRP connection to reader

[java] 303 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
IMPINJ_ENABLE_EXTENSIONS ...

[java] 354 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
IMPINJ_ENABLE_EXTENSIONS was successful

[java] 354 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
SET_READER_CONFIG with factory default ...

[java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
2011-03-16T13:06:52.734876-04:00 RSSI: -41

[java] 566 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
SET_READER_CONFIG Factory Default was successful

[java] 566 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
Building ADD_ROSPEC message from scratch ...

[java] 583 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
Sending ADD_ROSPEC message ...

[java] 609 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
ADD_ROSPEC was successful

[java] 610 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
ENABLE_ROSPEC ...

[java] 634 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
ENABLE_ROSPEC was successful

[java] 634 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
START_ROSPEC ...

[java] 655 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
START_ROSPEC was successful
```

```
  [java] EPC: 330833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:53.039862-04:00 RSSI: -37

  [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:53.100989-04:00 RSSI: -40

  [java] EPC: 300833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:53.550892-04:00 RSSI: -62

  [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:53.655947-04:00 RSSI: -37

  [java] EPC: 330833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:54.000079-04:00 RSSI: -40

  [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:54.224531-04:00 RSSI: -36

  [java] EPC: 350833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:54.777975-04:00 RSSI: -40

  [java] EPC: 300833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:54.842776-04:00 RSSI: -60

  [java] EPC: 330833b2ddd906c000000000 Antenna: 1 FirstSeen:
  2011-03-16T13:06:54.978229-04:00 RSSI: -40

  [java] 3014 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
  STOP_ROSPEC ...

  [java] 3031 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
  STOP_ROSPEC was successful

  [java] 3080 [main] INFO org.impinj.llrp.ltk.examples.docsample1 -
  CLOSE_CONNECTION was successful

BUILD SUCCESSFUL

Total time: 5 seconds
```

```
[revolution@RevolutionEDK examples]$
```

Figure 4.11 LTKJava – Linux Sample Execution

Run other samples in a similar fashion.

5 LTK Language Details

This section describes programming details for each of the Impinj LTK supported languages.

5.1 LTKCPP (LTKC)

This section describes the LTKCPP programming architecture. The structure of the LTKC and LTKCPP libraries are very similar. This section uses nomenclature for LTKCPP, but the organization and structure also applies to LTKC.

Applications interact with the LTKCPP primarily via the core classes: **CConnection, CErrorDetails**, and the **CElement** class and its children, **CParameter**, and **CMessage**.

5.1.1 Connection Class

LTK **CConnection** class manages the connection to the Reader, and also provides members to send messages, receive messages, perform transactions (a send with an expected reply), and to get error status. Examples of the use of the **CConnection** class are shown in Section 6.

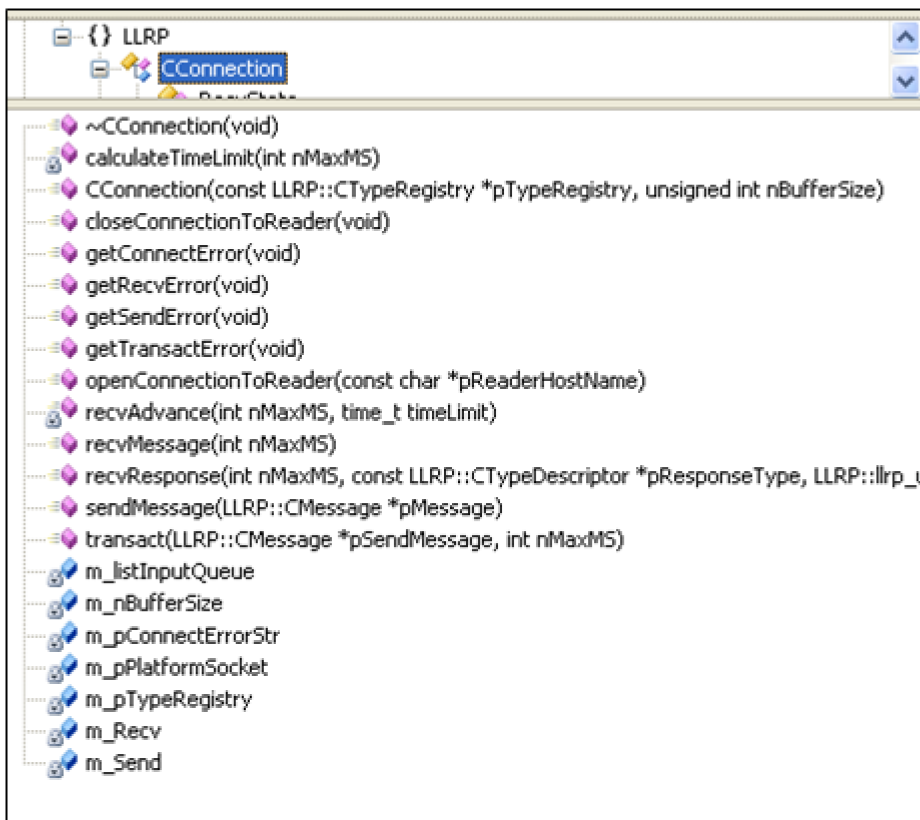


Figure 5.1 LTKCPP CConnection Class

5.1.2 Element Class

A **CElement** is a generic reference to an LLRP standard message, standard parameter, custom message, or custom parameter. All generated element classes derive from a common **CElement** base class.

The technique is necessarily different between C and C++. In C the base classes are encapsulated as the first member. In C++ normal class derivation is used.

Applications do not interact directly with a **CElement**. Applications build LLRP message and parameters using the **CMessage** and **CParameter** classes respectively.

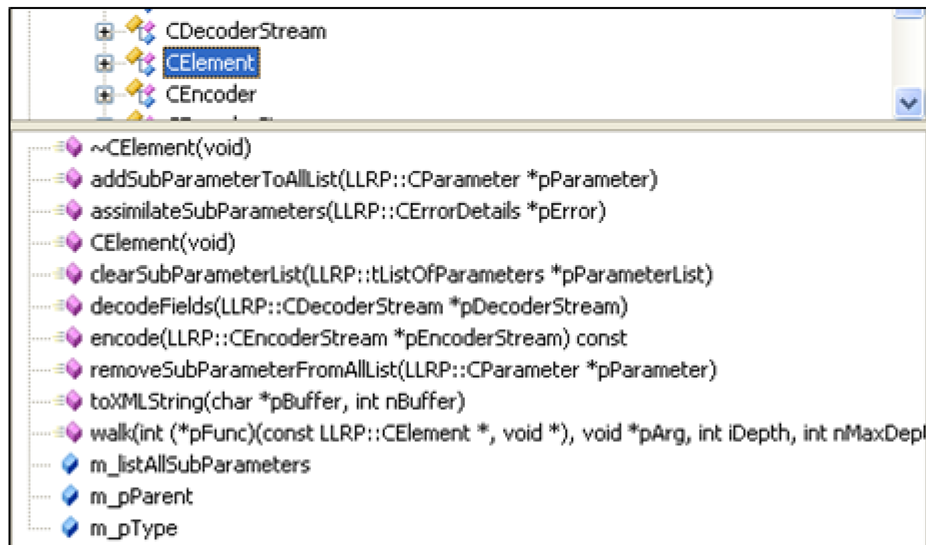


Figure 5.2 LTKCPP CElement Class

Element trees represent LLRP messages. The top-most element is an instance of a **CElement** class. It references zero or more sub-parameters. Each subparameter, which is also an element, might in turn reference zero or more subparameters, and so on. An element that has subparameters is said to be the enclosing **CElement** of each sub-parameter. For example, in Figure 5.3, element X is the enclosing element of A, B1, B2, B3, and C.

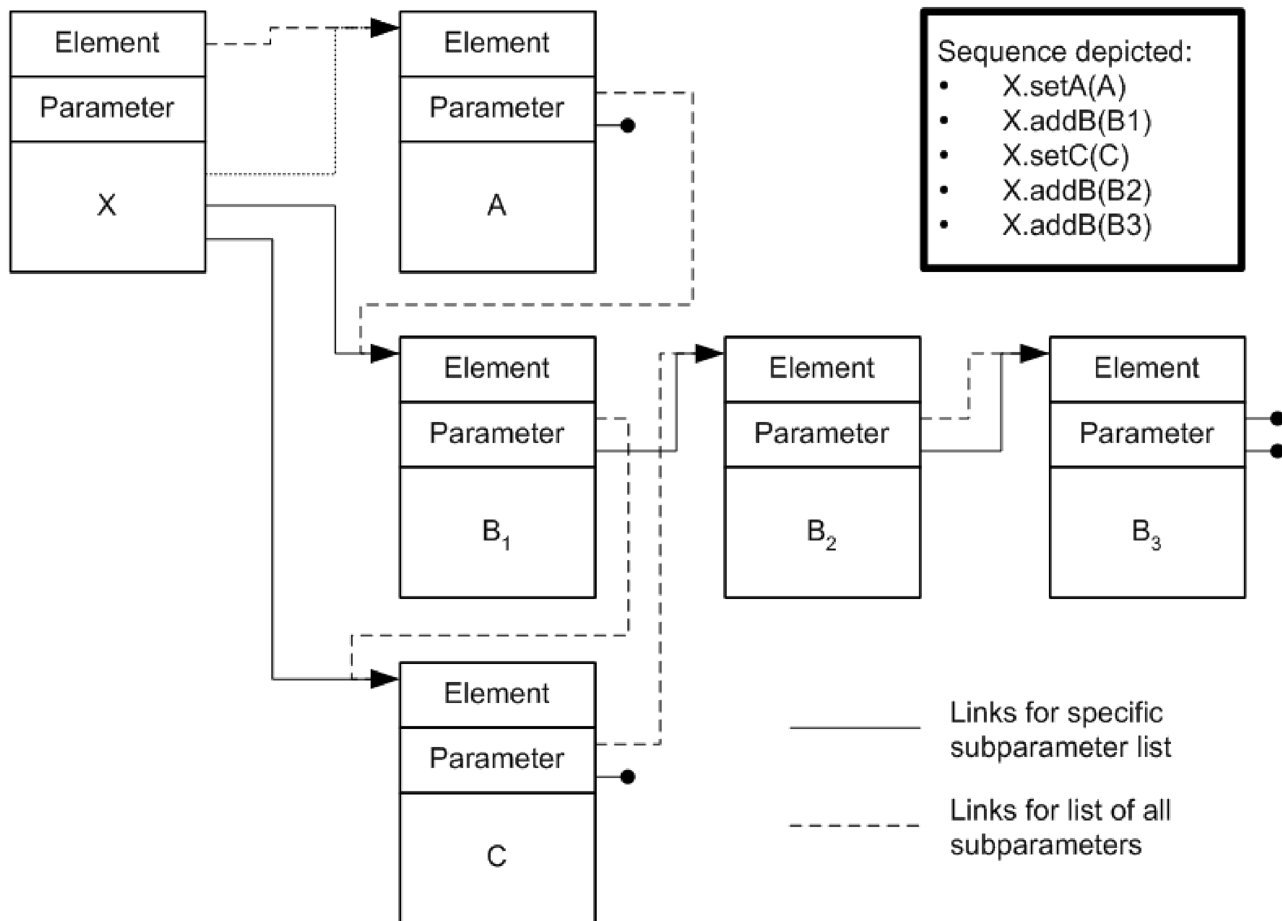


Figure 5.3 LTKCPP/LTKC Element Tree

Elements maintain an all-list, which is a list of all sub-parameters enclosed by the element. The all-list is important to allow correct functioning of LTKCPP. There is a separate list head within the element for each specific list of subparameters, such as the B list in Figure 5.3.

5.1.3 Fields, Subparameters, and Accessor Functions

A **CElement** might contain:

- Simple fields, such as integer or enumerated values. For these fields, there are the accessor functions ***getfield()*** and ***setfield()***. Vector fields, such as a string or a vector of 32-bit frequencies, have accessor functions ***getfield()*** and ***setfield()***. The ***setfield()*** accessor frees memory currently held by the vector field.
- Single subparameter pointers, used for both mandatory (1) and optional (0-1) subparameters. For subparameter pointers, use accessor functions ***getsubparam()*** and ***setsubparam().***

The ***setsubparam()*** accessor removes the current subparameter, if there is one,from the all-list and destructs it. It then adds the new subparameter to the all-list and sets the specific pointer. The ***setsubparam()*** function returns a status code, either OK (0) for success or a specific code for an error. The ***getsubparam()*** accessor returns a pointer to the subparameter. In most cases the type of the pointer matches the subparameter. However, for choices such as *OpSpecs* or *SpecParameter*, the pointer is a generic parameter type, so the application must determine the actual type and cast accordingly. For more information, see section 5.1.5 “How to Determine the Type of an Element”.

- Lists of subparameters, used for mandatory (1-N) and optional (0-N) lists. For these subparameters there are accessors beginsubparameter(), endsubparameter() for C++, nextsubparameter() for C, and addsubparameter(). The addsubparameter() accessor links the subparameter to the all-list. The addsubparam() function returns a status code, either OK (0) for success or a specific code for an error. C++ uses the Standard Template Library (STL) lists . In C there is no similar intermediate data structure. In both cases, similar to the single subparameter, the type of the pointer depends on whether the subparameter is a specific type or a choice.

In C++, the accessor functions are members of the class and the names are concise. In C, the names of the accessor functions can be long and include the name of the element type (such as ***LLRP_GET_READER_CAPABILITIES_getRequestedData***).

For example, the LTKCPP API for the LLRP *InventoryParameterSpec* contains the following set and get methods, including examples of lists (*antennaConfiguration*), fields (*ParameterSpecID*), and vectors (*ProtocolID*).

llrp_u16_t	**getInventoryParameterSpecID** (void) *Get accessor functions for the* ***LLRP*** *InventoryParameterSpecID field.*
void	**setInventoryParameterSpecID** (llrp_u16_t value) *Set accessor functions for the* ***LLRP*** *InventoryParameterSpecID field.*
EAirProtocols	**getProtocolID** (void) *Get accessor functions for the* ***LLRP*** *ProtocolID field.*
void	**setProtocolID** (**EAirProtocols** value) *Set accessor functions for the* ***LLRP*** *ProtocolID field.*
std::list< **CAntennaConfiguration** * >::iterator	**beginAntennaConfiguration** (void) *Returns the first element of the AntennaConfiguration sub-parameter list.*
std::list< **CAntennaConfiguration** * >::iterator	**endAntennaConfiguration** (void) *Returns the last element of the AntennaConfiguration sub-parameter list.*
void	**clearAntennaConfiguration** (void) *Clears the* ***LLRP*** *AntennaConfiguration sub-parameter list.*
EResultCode	**addAntennaConfiguration** (**CAntennaConfiguration** *pValue) *Add a AntennaConfiguration to the* ***LLRP*** *sub-parameter list.*
std::list< **CParameter** * >::iterator	**beginCustom** (void) *Returns the first element of the Custom sub-parameter list.*
std::list< **CParameter** * >::iterator	**endCustom** (void) *Returns the last element of the Custom sub-parameter list.*
void	**clearCustom** (void) *Clears the* ***LLRP*** *Custom sub-parameter list.*
EResultCode	**addCustom** (**CParameter** *pValue) *Add a Custom to the* ***LLRP*** *sub-parameter list.*

Figure 5.4 LTKCPP Accessor Methods

5.1.4 Why It Is Best to Use the Accessor Functions

As a rule, we have found it best to use element constructors that allocate instances, and use the accessor functions to manipulate and access fields and subparameters. In C++, this practice should be considered mandatory. In C, it is possible to use local instances (as shown in the example) by using great care.

We recommend this because of the destructors. LTKC/LTKCPP does not support mixing allocated elements with non-allocated element, such as local variable instances of elements. The destructors walk the all-lists recursively and delete (free) subparameters and vectors. Freeing a locally allocated instance can cause havoc.

The accessor functions handle list maintenance and incremental destructing of vectors and subparameters that are no longer needed. If you do not use the accessors, use appropriate memory

management in your application.

> **Impinj Best Practice**! *We strongly recommend using accessor methods to set and get fields and subparameters in the LTKCPP and LTKC.*

5.1.5 How to Determine the Type of an Element

At times a message or parameter type cannot be known in advance. The type displays and is determined at run-time. For example, a notification message arrives as an RO_ACCESS_REPORT, a READER_EVENT_NOTIFICATION, or a KEEPALIVE. Another example in the *ROSpecs* lists *SpecParameters*. Each of these might be an *AISpec*, an *RFSurvey* spec, or a *CustomParameter*. An additional example is *OpSpecs* and *OpSpecResults*.

All elements display a type label. The type label points to a type descriptor. You can determine the element type by comparing the type label pointer to the address of a known type descriptor. After the type is known, use casting for the specific element type.

This example shows the function ***awaitAndPrintReport()***. A LTKCPP sequence appears as:

LTKCPP

```
/*
* What happens depends on what kind of message
* received. Use the type label (m_pType) to
* discriminate message types.
*/

pType = pMessage->m_pType;

/*
* Is it a tag report? If so, print it out.
*/

if(&CRO_ACCESS_REPORT::s_typeDescriptor == pType)
{
   CRO_ACCESS_REPORT * pNtf;
   pNtf = (CRO_ACCESS_REPORT *) pMessage;
   printTagReportData(pNtf);
}
```

Figure 5.5 LTKCPP Sequence to Determine Element Type

5.1.6 Practical Information about Building Messages in LTKCPP

Build LTKCPP messages by using these working tips.

LTKCPP uses standard naming for all of its class names, methods, and members. After constructing the message by using the ***new*** operator, the programmer looks at all of the members with the name ***addXXX*** or ***setXXX***. Set all of the public LLRP parameters and fields through the LTK. Set functions that take a basic type do not require allocation of memory. Sub-parameter set functions that take a pointer as an argument require the sub-parameter to be created through another new operation. Add functions require that LLRP allows zero or more of these items within the containing message or parameter. One exception is ***addSubParameterToAllList***, which is an internal member function and is not used to set LLRP protocol data.

Parameters can be optional in LLRP.

The LTK does not provide any warnings or errors when mandatory sub-parameters are missing. The Reader returns these errors.

To build an LTKCPP message object using LTK-XML, follow these steps:

1. Create and validate an LTK-XML decoder using the type Registry that you built.

LTKCPP

```
CXMLTextDecoder * pDecoder;
/* Build a decoder to extract the message from XML */
pDecoder = new CXMLTextDecoder(m_pTypeRegistry,"setReaderConfig.xml");
if(NULL == pDecoder)
{
   /* Handle this error */
}
```

Figure 5.6 LTKCPP Constructing and LTK-XML Decoder

> **Impinj Best Practice**! *There are multiple constructors for importing XML: buffer, file, and libxml2 DOM. If you create a decoder using the DOM constructor, the application requires you to free the DOM tree memory afterwards.*

2. Decode the message.

LTKCPP

```
/* Decode the message */
CMessage * pCmdMsg;
pCmdMsg = pDecoder->decodeMessage();
if(NULL == pCmdMsg)
{
   /* Handle this error */
}
```

Figure 5.7 LTKCPP Decoding using an LTK-XML Decoder

3. Destroy the decoder (if you are not using it anymore)

LTKCPP

```
delete pDecoder;
```

Figure 5.8 LTKCPP Destroying an LTK-XML Decoder

4. To modify the message, convert it to the correct type. In this case, it is a SET_READER_CONFIG message. If you do not need to touch the message (for example, to modify a field), this step can be omitted.

LTKCPP

```
if(&CSET_READER_CONFIG::s_typeDescriptor != pCmdMsg->m_pType)
{
   /* Handle this error */
}

/* get the message as a SET_READER_CONFIG */
pCmd = (CSET_READER_CONFIG *) pCmdMsg;
```

Figure 5.9 LTKCPP Converting a Message to a Known Type

5.1.7 Error Reporting (ErrorDetails Class)

Error reporting is provided through the CErrorDetails structure, where LTKC/LTKCPP provides the following error information to the application:

- A status code, where 0 always represents OK. The status code is enumerated and the application easily distinguishes errors. Error codes are defined by an enumeration in 'ltkc_base.h' or 'ltkcpp_base.h'.
- A string error message, where NULL represents no error. This simplifies the application log or display error message without needing to interpret the status code.
- A pointer to a reference ***typeDescriptor*** contains basic information about an LLRP message or parameter type, such as name and type number. An application uses a ***typeDescriptor*** to print the name of a type. The meaning of the ***errorDetail'*s *typeDescriptor*** depends on the error. For example, this might be the type of an unexpected parameter, or a missing parameter, or a parameter being decoded when there was a data underrun. NULL represents no reference type available.
- A pointer to a reference ***fieldDescriptor*** contains basic information about an LLRP field, such as the name and basic type. An application uses a ***fieldDescriptor*** to print the name of a field. The ***errorDetail'*s *fieldDescriptor*** reports only field encode and decode errors. NULL represents no reference field available.

CErrorDetails instances appear within a decoder instance, encoder instance, and a connection instance. The application can access them directly.

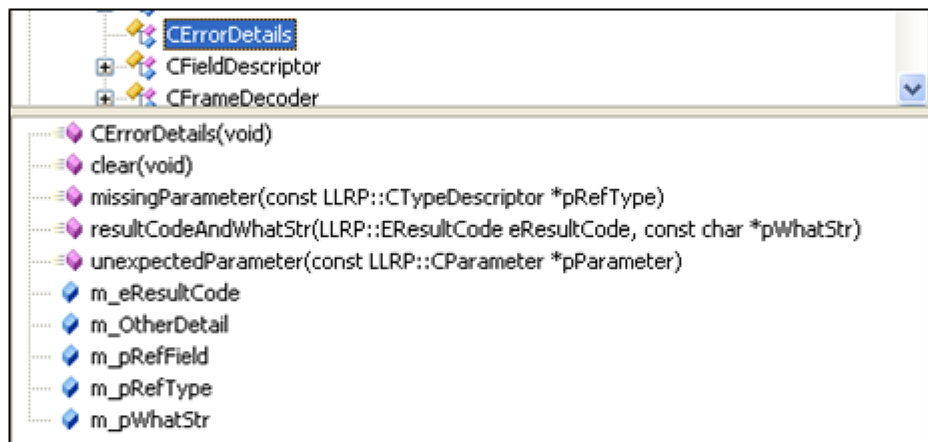


Figure 5.10 LTKCPP CErrorDetails Class

In the example program, see the ***transact()*** function, which is a portion that detects an error and then prints the available error detail information.

LTKCPP

```
if(NULL == pRspMsg)
{
   const CErrorDetails * pError = pConn->getTransactError();
   printf("ERROR: %s transact failed, %s\\n",
   pSendMsg->m_pType->m_pName,
   pError->m_pWhatStr ? pError->m_pWhatStr : "no reason given");
   if(NULL != pError->m_pRefType)
   {
      printf("ERROR: ... reference type %s\\n",
      pError->m_pRefType->m_pName);
   }

   if(NULL != pError->m_pRefField)
   {
      printf("ERROR: ... reference field %s\\n",
      pError->m_pRefField->m_pName);
   }

   return NULL;
}
```

Figure 5.11 LTKCPP– Error Reporting Example

5.1.8 Extensions

This section provides a brief description of extensions and extension points.

LLRP Model of Extensions

The LLRP specification defines a set of standard messages and parameters. It also defines a CUSTOM_MESSAGE and a *customParameter*, which are intended to encapsulate vendor specific extensions. Some of the standard messages and parameters include extension points represented as 0-N *customParameters*.

The LLRP specification states that when an application receives a *customParameter* that it does not recognize, it must tolerate the parameter. The standards group decision uses analysis of the LLRP use cases. For example, consider GET_READER_CAPABILITIES_RESPONSE. It might contain custom parameters within reader capabilities that are vendor-specific. The application must be capable of safely ignoring custom parameters if the connected reader is unknown.

However, the LLRP specification says that when a Reader receives a custom parameter that it does not recognize, it must report an error. This also derives from the standards group decisions

for the LLRP use cases. For example, assume that the application sends a custom parameter in a SET_READER_CONFIGURATION that the Reader does not understand. It is a safe conclusion that the Reader functions will not act as the application intends. Normally, the Reader responds with an error.

Therefore, the LTK implementations, LTKC/LTKCPP included, provide for tolerance of unrecognized custom parameters and leave it to the higher layers to detect and handle errors.

Constructing Custom Elements

When an extension definition processes, the same templates are used to create per-element classes (structs), type descriptors, constructor functions, destructor functions, and so on.

This means that constructing and manipulating a custom message or parameter is similar to the process for a standard message or a parameter. The primary difference is that standard elements have specific accessors for standard subparameters. Custom parameters use the enclosing element accessors for a single list of all custom elements, rather than a specific list for each custom parameter type.

Accessors

Standard parameters and messages with extension points represent them as a list of *CustomParameters*. When *CustomParameters* occur within a parameter or message, the LTKCPP code generation process creates a list of CParameters (similar to choices), and names the subparameter list **Custom**. The accessor functions for the custom list use pointers to the generic CParameters type. For more information, see section 5.1.3 Fields, Subparameters, and Accessor Functions.

If the application tries to add a *CustomParameter* to an extension point list (also known as Custom) that is not allowed at that point, the ***addCustom()*** accessor function returns an error. See Figure 5.4 for an example of the custom parameter accessors.

When the application scans the extension point list (also known as *CustomParameter*), it must check each subparameter type. Before you (optionally) process the subparameters, see section 5.1.5 How to Determine the Type of an Element for more information.

Enrollment in TypeRegistry

Each LTK definition file processes the LTKCPP enrollment function when compiling the libraries. The enrollment function takes a ***typeRegistry*** as an argument, and adds the ***typeDescriptors*** for the definition file to the registry. The application program must call the ***enrollment*** function for each set of extensions it uses.

In this example, function ***run()*** calls ***getTheTypeRegistry().*** This function constructs a new CTypeRegistry and enrolls the standard LLRP message and parameter types. See Section 6.2 for an example of this process.

5.2 LTKNET

The LTKNET follows the LLRP class hierarchy. A base message class implements the LLRP message header information, as well as the encoding/decoding method **FromBitArray**, **ToBitArray,** and the XML encoding method **ToString.** All LLRP messages (including CUSTOM_MESSAGE) are derived from this class. A base parameter class implements the LLRP parameter header information, as well as the encoding/decoding method **FromBitArray**, **ToBitArray**, and the XML encoding method **ToString**. All LLRP parameters (including CUSTOM_PARAMETERS) derive from this class.

5.2.1 LLRPClient

The LLRPClient class controls the state for the connection to a single Reader. This class contains methods to open and close the Reader connection, which is a transaction method that sends LLRP messages and awaits the responses. Event handlers handle asynchronous incoming messages from the Reader.

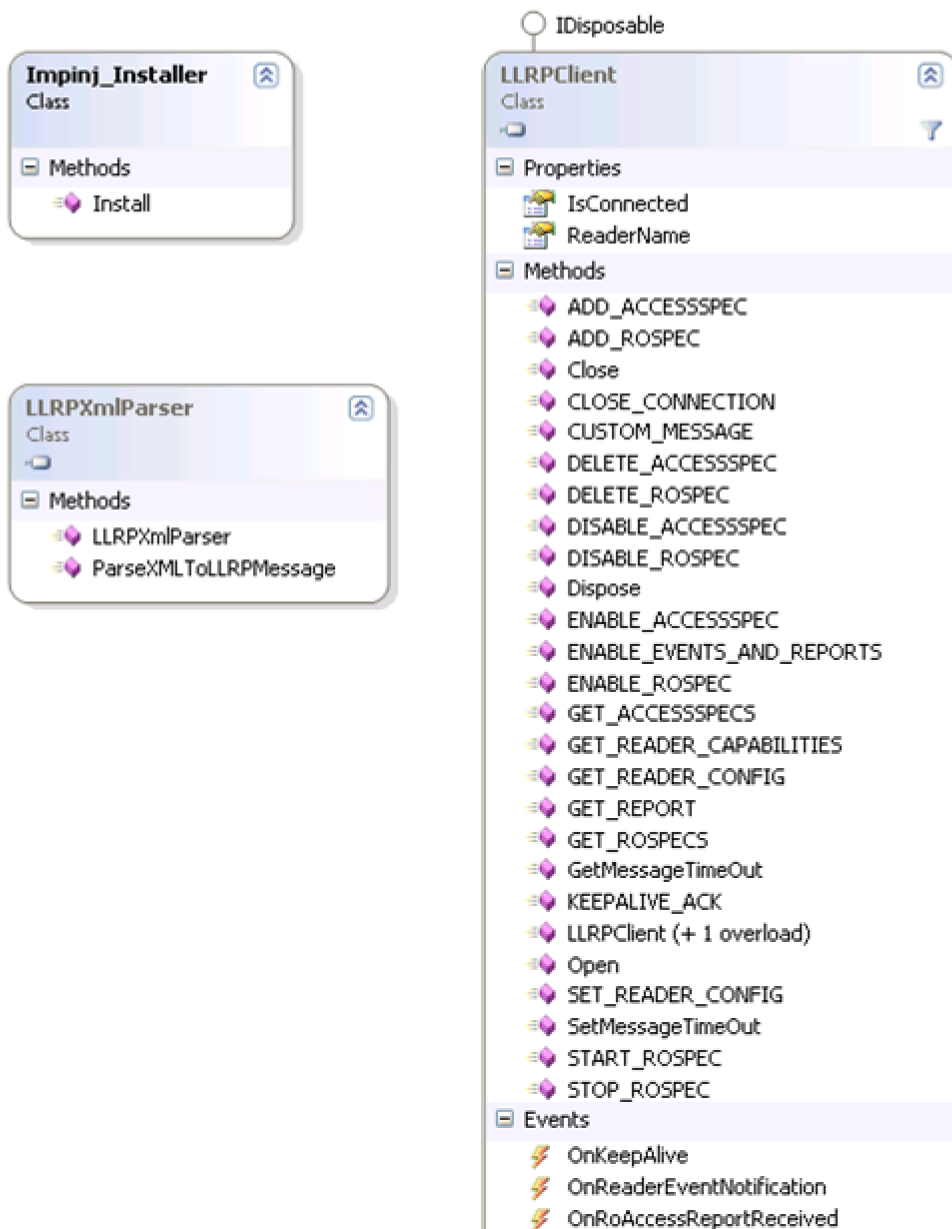


Figure 5.12 LTKNET – Core Application Classes

5.2.2 Impinj Installer Class

The **Impinj_Installer** class supplies a single static method that allows the application to install the Impinj Extensions. The **LLRPXMLParser** class allows the LTKNET to import LTK-XML messages directly into C# objects. See Figure 5.12 for details.

5.2.3 LLRP Message Classes

Figure 5.13 shows a class diagram with select message types. Many messages that enable/disable, start/stop, or get data from the Reader all follow the same format and are not shown here. The following primary message classes are required to perform basic LLRP operations.

Figure 5.13 LTKNET – Core Message Classes

5.2.4 Handling ROSpecs

Key classes and methods for building *PARAM_ROSpecs* are shown in Figure 5.14. Note that the *PARAM_AISpec* is included within the *ROSpec* as part of its **SpecParameter** list. This coupling is loose because LLRP allows coupling of other specs into the **PARAM_ROSpecs** as well. Speedway supports only *AISpecs* as part of the *PARAM_ROSpecs SpecParameter* list.

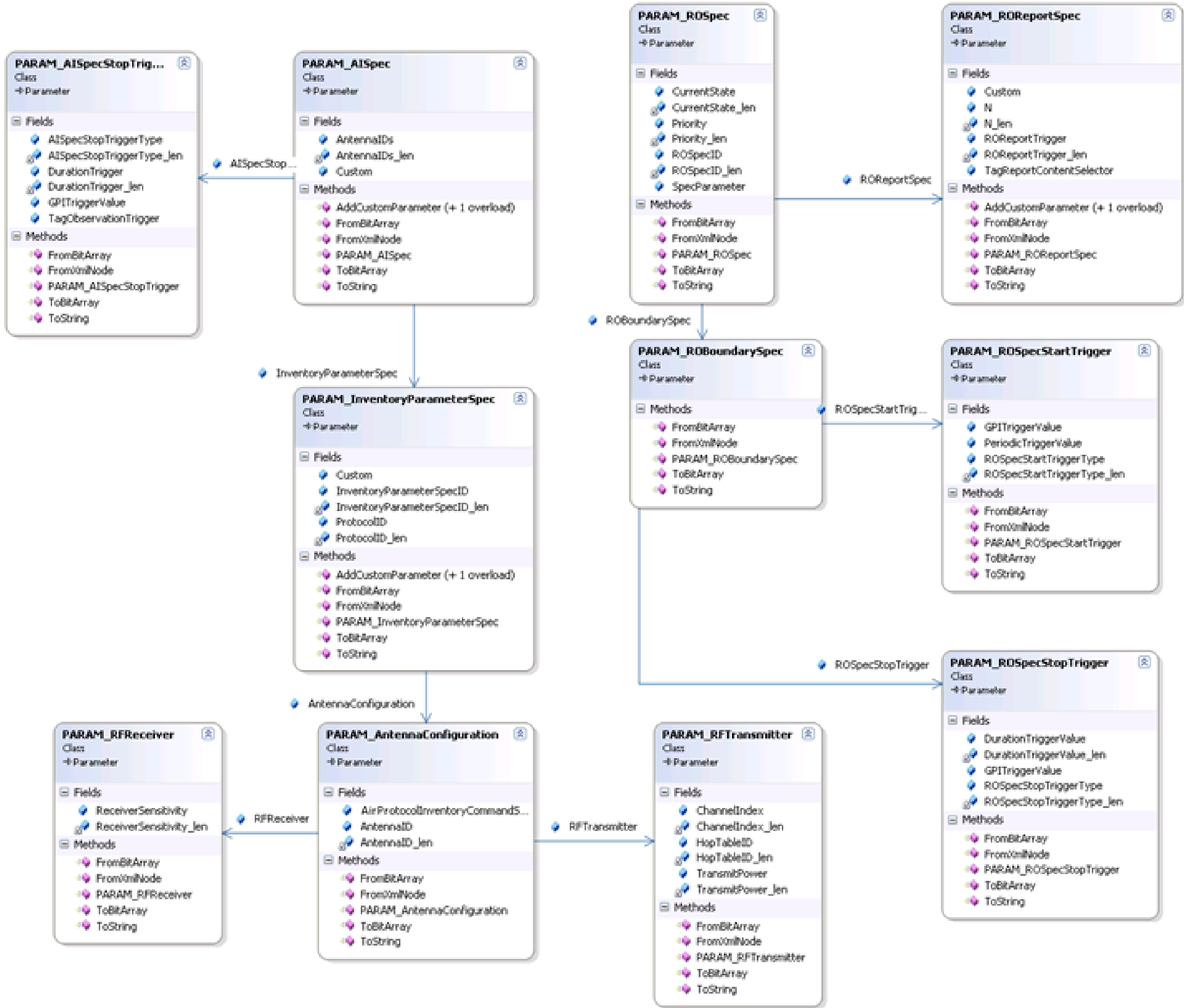


Figure 5.14 LTKNET – Core RoSpec Classes

5.2.5 Handling AccessSpecs

Key parameters for building a *PARAM_AccessSpec* are shown in Figure 5.15. The individual Gen2 operations (read, write ...) loosely couple into the access command object through the *UNION_AccessCommandOpSpec* union. This allows the expansion of non-gen2 based operations as per the LLRP standard charter. The same is true of the *PARAM_C1G2TagSpec* within the *UNION_AirProtocolTagSpecs*.

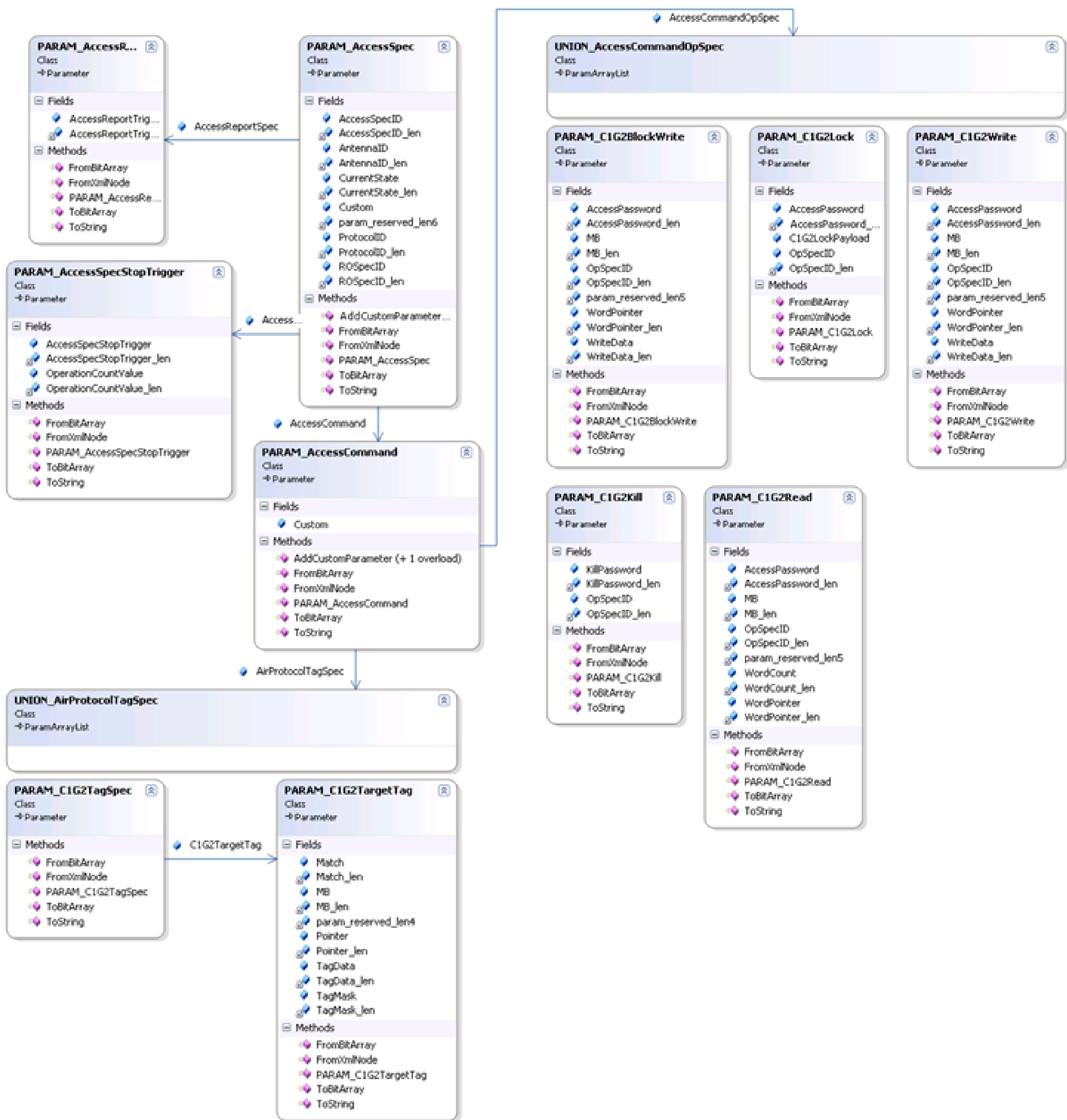


Figure 5.15 LTKNET – Core AccessSpec Classes

5.3 LTKJAVA

The following sections explain specifics of using LTKJAVA for encoding, decoding, sending, and receiving LLRP messages.

5.3.1 Receiving LLRP Messages

The most basic interface to the Reader is via the ***LLRPEndpoint*** interface. This interface provides a mechanism to receive messages from the Reader. User classes that interact with the Reader should implement this interface.

Messages from the Reader are passed to the application through the ***messageReceived*** method of the **LLRPEndpoint** class. From a simple application perspective, implementing this ***messageReceived*** method gets the EPC tag data from the Reader. The example in Figure 5.16 shows a simple implementation of the ***messageReceived*** method.

LTKJAVA

```
public void messageReceived(LLRPMessage message) {
   // convert all messages received to LTK-XML representation
   // and print them to the console
   logger.debug("Received " + message.getName() +
      " message asynchronously");

   if (message.getTypeNum() == RO_ACCESS_REPORT.TYPENUM) {
      RO_ACCESS_REPORT report = (RO_ACCESS_REPORT) message;
      List<TagReportData> tdlist = report.getTagReportDataList();

      for (TagReportData tr : tdlist) {
         logOneTagReport(tr);
      }

   } else if (message.getTypeNum() ==
            READER_EVENT_NOTIFICATION.TYPENUM) {
      // Insert Logic for READER_EVENT_NOTIFICATION here
   }
}
```

Figure 5.16 LTKJAVA – Implementing LLRPEndpoint MessageReceived

5.3.2 Connections

LLRPConnection objects hold the state for a connection to a single Reader. A connection allows the sending and receiving of **LLRPMessage**(s).

A connection self-initiates, or the Reader initiates the connection. Self-initiated connections open (initiate) by the application. Typical applications use self-initiated connections.

Although you can create an **LLRPConnection** object directly, most applications use a subclass of **LLRPConnection**, which creates their connection objects. For self-initiated connections, use the **LLRPConnector** class (derived from **LLRPConnection**), which creates an **LLRPConnection** object. To use Reader-initiated connections, create an **LLRPAcceptor** object, which is also derived from **LLRPConnection**.

Figure 5.17 shows an example of creating a connection to a Reader with the **LLRPConnector** class. This example shows that connection failures are returned to the application through the ***LLRPConnectionAttemptFailedException*** event.

LTKJAVA

```
connection = new LLRPConnector(this, ip);

// connect to reader
// LLRPConnector.connect waits for successful
// READER_EVENT_NOTIFICATION from reader
try
{
   logger.info("Initiate LLRP connection to reader");
   ((LLRPConnector) connection).connect();
}
catch (LLRPConnectionAttemptFailedException e1)
{
   e1.printStackTrace();
   System.exit(1);
}
```

Figure 5.17 LTKJAVA – Application Initiated Reader Connection

5.3.3 Sending and Receiving Messages

After it is connected, the application sends and receives messages through the **LLRPConnection** object (parent) returned by the **LLRPConnector** constructor. The **LLRPConnection** API provides two different methods when it is sending messages: send and transact. Send delivers the

message to the Reader; as soon as the message is sent, the method returns. It does not guarantee that the message is received by the Reader. Transact sends the message and then awaits a response from the Reader. The method blocks until the response is received or until a timeout occurs. Transact should be used for most LLRP messages sent from the application. An exception is the **GET_REPORT** message, which does not have a response message.

5.3.4 LLRPMessage and LLRPParameter

In the previous few sections, we reviewed how to communicate with the Reader and to how send and receive **LLRPMessage** objects. This section discusses **LLRPMessages** and their definitions.

LLRPMessage(s) send and receive information to and from the Reader. An LLRP contains zero or more simple data fields followed by a set of **LLRPParameters**.

LLRP Messages all derive from **LLRPMessage** classes. This provides the basic message interface and encodes LTKJAVA objects into LLRP Binary or LTK-XML.

5.3.5 Building LLRP Messages

There are two basic methods you can use to build **LLRPMessage** objects. First, they can be constructed by creating the individual parameters and adding them to the message using Java methods. Second, XML strings or files can also create LLRP messages.

Constructing simple messages by using LTKJAVA objects is typically the easiest method. Figure 5.18 shows an example of constructing an **ENABLE_ROSPEC** message from objects. The object creates and sets the single field (***ROSpecID***).

LTKJAVA

```
logger.info("ENABLE_ROSPEC ...");
ENABLE_ROSPEC ena = new ENABLE_ROSPEC();
ena.setMessageID(getUniqueMessageID());
ena.setROSpecID(rospec.getROSpecID());
```

Figure 5.18 LTKJAVA – Constructing an LLRP Message from LTKJAVA Objects

Sometimes it is more convenient to construct longer messages from LTK-XML files. For a description of how to use LTK-XML, see Section 3.5. The code in Figure 5.19 shows an example of constructing an ADD_ROSPEC message from an XML file.

LTKJAVA

```java
private ADD_ROSPEC buildROSpecFromFile()
{
   logger.info("Loading ADD_ROSPEC message from file ADD_ROSPEC.xml...");
   try
   {
      LLRPMessage addRospec = Util.loadXMLLLRPMessage(new File(
        "./source/org/impinj/llrp/ltk/examples/docsample1/ADD_ROSPEC.xml"));
      // TODO make sure this is an ADD_ROSPEC message
      return (ADD_ROSPEC) addRospec;
   }
   catch (FileNotFoundException ex)
   {
      logger.error("Could not find file");
      System.exit(1);
   }
   catch (IOException ex)
   {
      logger.error("IO Exception on file");
      System.exit(1);
   }
   catch (JDOMException ex)
   {
      logger.error("Enable to convert LTK-XML to DOM");
      System.exit(1);
   }
   catch (InvalidLLRPMessageException ex)
   {
      logger.error("Enable to convert LTK-XML to Internal Object");
      System.exit(1);
   }

   return null;
}
```

Figure 5.19 LTKJAVA – Constructing an LLRP Message from LTK-XML

Note the four exceptions that this routine catches, including:

- The JDOM exception indicates that the XML file could not be read properly into a DOM. If this occurs, check for syntax errors or other XML errors.

- The **InvalidLLRPMessageException** indicates that the LTK-XML message could not be properly converted to LTKJAVA objects due to its format or contents. If this occurs, double-check namespaces and mandatory LLRP parameters for errors.
- IOException and FileNotFoundException are standard Java file I/O exceptions.

5.3.6 Custom Extensions

Impinj custom extensions integrate into the LTKJAVA library. LLRP allows custom extensions at "extension points". The LTK supports extension points as lists of custom parameters. If you see a ***getCustomList*** method for a message or parameter, the message may contain custom extensions.

To learn what Impinj extensions might be present in each parameter or message, see the Octane LLRP reference guide. The example in Figure 5.20 iterates a custom extension list.

LTKJAVA

```
List<Custom> clist = report.getCustomList();
for (Custom cust : clist)
{
   logOneCustom(cust);
}
```

Figure 5.20 LTKJAVA – Iterating Custom Sub-Parameters

When iterating this list, the application must validate the type of the custom extension before it processes the contents and casts the *Custom* parameter back to the appropriate type. See Figure 5.21 for an example of this. Note that the vendor ID and subtype are checks against the custom extension.

LTKJAVA

```
private void logOneCustom(Custom cust)
{
   String output = "";
   if(!cust.getVendorIdentifier().equals(25882))
   {
      logger.error("Non Impinj Extension Found in message");
      return;
   }

   if (cust.getParameterSubtype().equals(
            ImpinjTagInformation.PARAMETER_SUBTYPE))
```

```
    {
        ImpinjTagInformation itag = (ImpinjTagInformation) cust;
        /* process this */
    }
}
```

Figure 5.21 LTKJAVA – Processing Custom Sub-Parameter

5.3.7 Logging

LTKJAVA uses the de facto standard log4j interface. See the examples in the next section for instances of using log4j in LTKJAVA applications. For more information about using log4j, see http://logging.apache.org/log4j/1.2/index.html.

5.3.8 Summary

The following shows the key elements of using the LTKJava:

- Implements the **LLRPEndpoint** interface in your class.
- Implements the ***messageReceived*** method to get asynchronous tag data.
- Creates an **LLRPConnection** within your class.
- Sends messages to the Reader via the send or transact methods.
- Builds a message using their LTKJAVA classes and subparameter classes.
- Decodes message using their LTKJAVA classes and subparameter classes.

6 Adding the LTK into your programs

6.1 Compiling and Linking

LTKC – add the LTKC to your Linux programs by specifying the **'libltkc'** directory in your compiler's include path and link path. Include both the **'ltkc__xxx.a'** and **'ltkcimpinj__xxx.a'** files (xxx is either x86, arm, or xScale).

LTKC

```
LTKC_DIR = ..

LTKC_LIBS = $(LTKC_DIR)/libltkc_x86.a

$(LTKC_DIR)/libltkcimpinj_x86.a

LTKC_INCL = -I$(LTKC_DIR)

CFLAGS = -g $(LTKC_INCL)

example1 : example1.c

   $(CC) $(CFLAGS) -o example1 example1.c $(LTKC_LIBS)
```

Figure 6.1 LTKC – Compiling and Linking

A conversion from LTK-XML to LTKC objects requires libxml2__xxx.a (where xxx is either x86, arm, or xScale), as shown in Figure 6.2.

LTKC

```
LIBXML2_LIB = $(LTKC_DIR)/libxml2$(SUFFIX).a

example1 : example1.c

   $(CC) $(CFLAGS) -o example1 example1.c $(LTKC_LIBS) $(LIBXML2_LIB)
```

Figure 6.2 LTKC – Compiling and Linking with Libxml2

LTKCPP – To add the LTKCPP to your Linux programs, specify the libltkcpp directory in your compiler's include path and link path. You must include both the **'ltkcpp__ <target>.a'** and **'ltkcppimpinj__<target>.a'** files **(<target>** is either x86, or xscale).

LTKCPP

```
LTKCPP_DIR = ..

LTKCPP_LIBS = $(LTKCPP_DIR)/libltkcpp_x86.a \\
              $(LTKCPP_DIR)/libltkcppimpinj_x86.a
LTKCPP_INCL = -I$(LTKCPP_DIR)
CPPFLAGS = -g $(LTKCPP_INCL)
example1 : example1.cpp
   $(CXX) $(CPPFLAGS) -o example1 example1.cpp $(LTKCPP_LIBS)
```

Figure 6.3 LTKCPP – Compiling and Linking

Converting from LTK-XML to LTKCPP objects also requires libxml2_xxx.a (xxx is either x86, arm, or xScale) as shown in Figure 6.4.

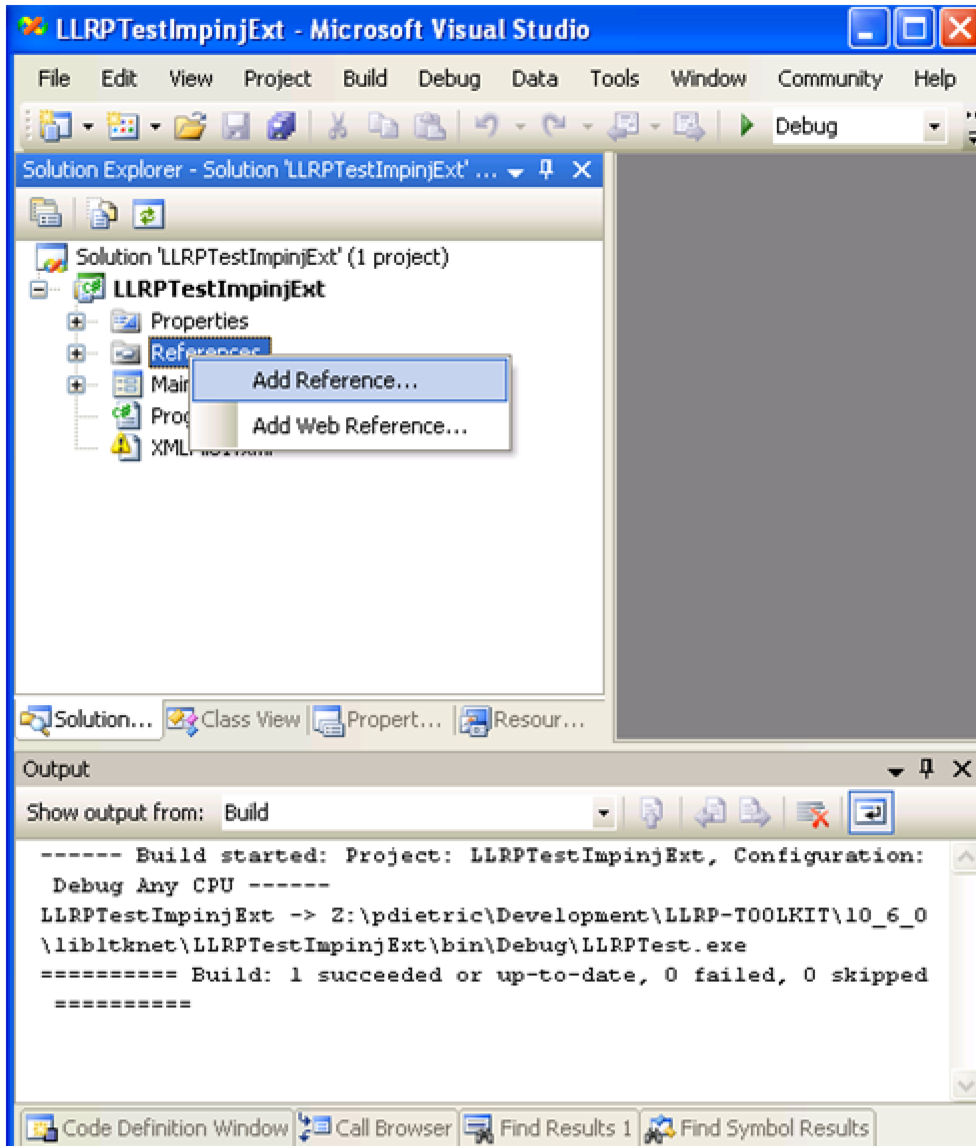
LTKCPP

```
LIBXML2_LIB = $(LTKC_DIR)/libxml2$(SUFFIX).a
example1 : example1.c
   $(CCP) $(CPPFLAGS) -o example1 example1.c $(LTKCPP_LIBS) $(LIBXML2_LIB)
```

Figure 6.4 LTKCPP – Compiling and Linking with Libxml2

LTKNET – To add the LTKNET to your C# .NET programs, add **'LLRP.dll'** and **'LLRP.Impinj.dll'** as references to your project. using this procedure.

1. Right click **References**, and then click **Add reference**.


LLRPTestImpinjExt - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Solution Explorer - Solution 'LLRPTestImpinjExt' ...
Solution 'LLRPTestImpinjExt' (1 project)
LLRPTestImpinjExt
Properties
Add Reference...
Add Web Reference...
Solution... Class View Propert... Resour...
Output
Show output from: Build
------ Build started: Project: LLRPTestImpinjExt, Configuration: Debug Any CPU ------
LLRPTestImpinjExt -> Z:\pdietric\Development\LLRP-TOOLKIT\10_6_0\libltknet\LLRPTestImpinjExt\bin\Debug\LLRPTest.exe
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
Code Definition Window Call Browser Find Results 1 Find Symbol Results

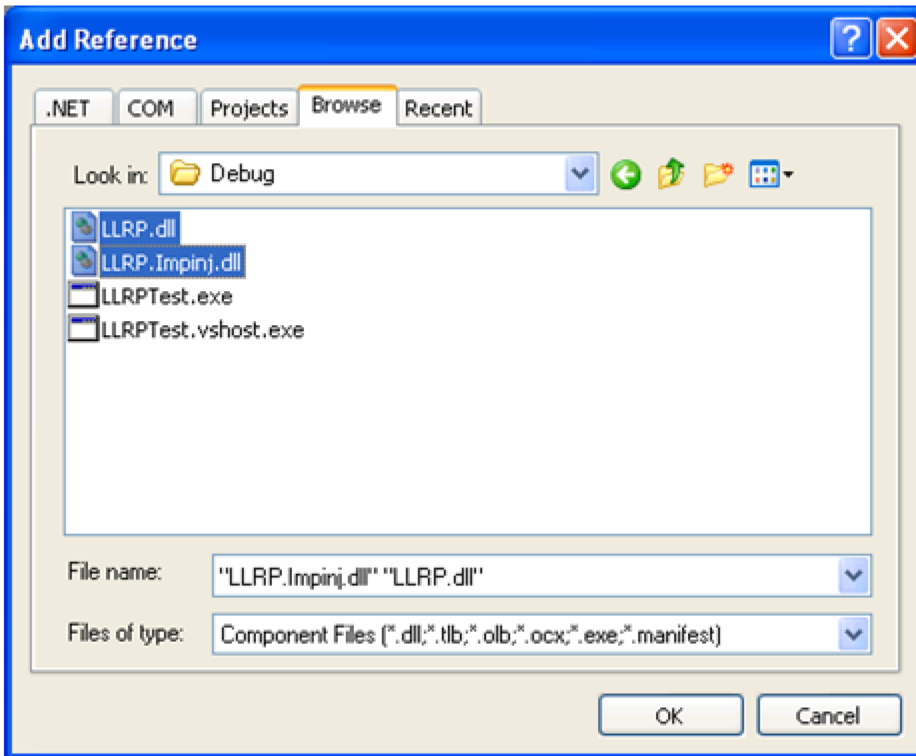


Figure 6.5 LTKNET – Compiling and Linking

LTKJava – To add LTKJava to your applications, add the JAR to the classpath, and then add the import statements required to use the library. A sample import set is shown in Figure 6.6.

LTKJAVA

```
import org.llrp.ltk.exceptions.*;
import org.llrp.ltk.generated.enumerations.*;
import org.llrp.ltk.generated.interfaces.*;
import org.llrp.ltk.generated.messages.*;
import org.llrp.ltk.generated.parameters.*;
import org.llrp.ltk.types.*;
import org.llrp.ltk.net.LLRPConnection;
import org.llrp.ltk.net.LLRPConnectionAttemptFailedException;
import org.llrp.ltk.net.LLRPConnector;
import org.llrp.ltk.net.LLRPEndpoint;
import org.llrp.ltk.util.Util;
```

Figure 6.6 LTKJava – Import Statement Example

6.2 Initializing the library

Since the LTK libraries are based on a set of compiled XML description files, a dynamic packet registry must be created in some libraries for decoding packets from the base LLRP, and from the Impinj Octane extensions. The following examples show how to correctly initialize the LTK and register, which allows the LTK to send and receive Impinj Octane messages and parameters.

LTKCPP

```
CTypeRegistry * pTypeRegistry;
/*
* Allocate the type registry. This is needed
* by the connection to decode.
*/
pTypeRegistry = getTheTypeRegistry();

if(NULL == pTypeRegistry)
{
   printf("ERROR: getTheTypeRegistry failed\\n");
   return -1;
}

/*
**Impinj Best Practice!**
*Enroll impinj extension types into the*
*type registry, in praparation for using*
*Impinj extension params.*
*/
LLRP::enrollImpinjTypesIntoRegistry(pTypeRegistry);
```

Figure 6.7 LTKCPP – Initializing

LTKNET

```
#region Initializing
  {
     Console.WriteLine("Initializing\\n");
     //Create an instance of LLRP reader client.
     reader = new LLRPClient();
     //**Impinj Best Practice!** *Always install the Impinj extensions*
     Impinj_Installer.Install();
```

```
  }
#endregion
```

Figure 6.8 LTKNET – Initializing

Note: LTKJAVA registry builds at compile time; this requires no special registration of Impinj Octane messages.

LTKJAVA

```
// create client-initiated LLRP connection
LLRPConnector connection = new LLRPConnector (this, READER_IP_ADDRESS);
```

Figure 6.9 LTKJAVA – Initializing

6.3 Connecting to the Reader

To establish a connection with the Reader, the application first creates a connection object. After creating the connection object, a connection with the Reader initiates. Once the Reader accepts the connection, it always sends a READER_EVENT_NOTIFICATION that contains the *ConnectionStatusEvent.* The standard mandates that the client application must wait for this notification before continuing Reader communications.

6.3.1 LTKCPP

LTKCPP requires that you create connection objects. The connection object needs a buffer to serialize outgoing messages and to de-serialize incoming messages. Select this buffer based on your application's use of LLRP. Choose the maximum buffer size based on the size of the reports you expect to receive from the Reader. If you are reporting data at the end of very long specs, and with large tag populations, choose your buffer size carefully. However, if you are working with immediate reporting (or reports of only a few hundred tags), a 32K buffer size is sufficient. Figure 6.10 shows how to create a connection object.

LTKCPP

```
/*
* Construct a connection (LLRP::CConnection).
* Using a 32kb max frame size for send/recv.
* The connection object is ready for business
* but not actually connected to the reader yet.
```

```
*/

pConn = new CConnection(pTypeRegistry, 32u*1024u);
if(NULL == pConn)
{
   printf("ERROR: new CConnection failed\\n");
   return -2;
}

/*
* Open the connection to the reader
*/

if(m_Verbose)
{
   printf("INFO: Connecting to %s....\\n", pReaderHostName);
}

rc = pConn->openConnectionToReader(pReaderHostName);
if(0 != rc)
{
   printf("ERROR: connect: %s (%d)\\n", pConn->getConnectError(), rc);
   delete pConn;
   return -3;
}
/*
* Record the pointer to the connection object so other
* routines can use it.
*/
m_pConnectionToReader = pConn;
if(m_Verbose)
{
   printf("INFO: Connected, checking status....\\n");
}
```

Figure 6.10 LTKCPP – Creating Connection

In LTKCPP, the library does not handle the *ConnectionStatusEvent* automatically. Special code is required to verify that the connection was accepted properly. Figure 6.11 shows how to wait for this event:

LTKCPP

```
CMessage * pMessage;
CREADER_EVENT_NOTIFICATION *pNtf;
CReaderEventNotificationData *pNtfData;
CConnectionAttemptEvent * pEvent;

/*
* Expect the notification within 10 seconds.
* It is suppose to be the very first message sent.
*/
pMessage = recvMessage(10000);

/*
* recvMessage() returns NULL if something went wrong.
*/
if(NULL == pMessage)
{
   /* recvMessage already tattled */
   goto fail;
}

/*
* Check to make sure the message is of the right type.
* The type label (pointer) in the message should be
* the type descriptor for READER_EVENT_NOTIFICATION.
*/
if(&CREADER_EVENT_NOTIFICATION::s_typeDescriptor != pMessage->m_pType)
{
   goto fail;
}

/*
* Now that we are sure it is a READER_EVENT_NOTIFICATION,
* traverse to the ReaderEventNotificationData parameter.
*/

pNtf = (CREADER_EVENT_NOTIFICATION *) pMessage;
pNtfData = pNtf->getReaderEventNotificationData();
if(NULL == pNtfData)
{
   goto fail;
```

```
}
/*

* The ConnectionAttemptEvent parameter must be present.
*/
pEvent = pNtfData->getConnectionAttemptEvent();
if(NULL == pEvent)
{
   goto fail;
}

/*
* The status in the ConnectionAttemptEvent parameter
* must indicate connection success.
*/
if(ConnectionAttemptStatusType_Success != pEvent->getStatus())
{
   goto fail;
}

/*
* Done with the message
*/
delete pMessage;
/*
* Victory.
*/
return 0;

fail:

/*
* Something went wrong. Tattle. Clean up. Return error.
*/
printf("ERROR: checkConnectionStatus failed\\n");
delete pMessage;
return -1;
```

Figure 6.11 LTKCPP – Validating Connection

6.3.2 LTKNET

LTKNET hides the packet encoding/decoding registry from the API. LTKNET also automatically awaits the *ConnectionStatusEvent* event and reports back with the status through the ***reader.Open()*** command. It is important to check the status as well as the command return-code.

LTKNET

```
#region Connecting
{
   Console.WriteLine("Connecting To Reader\\n");
   ENUM_ConnectionAttemptStatusType status;

   //Open the reader connection. Timeout after 5 seconds
   bool ret = reader.Open(readerName, 5000, out status);

   //Ensure that the open succeeded and the reader
   // returned the correct connection status result
   if (!ret \\ status != ENUM_ConnectionAttemptStatusType.Success)
   {
      Console.WriteLine("Failed to Connect to Reader \\n");
      return;
   }
}
#endregion
```

Figure 6.12 LTKNET – Creating/Validating Connection

6.3.3 LTKJAVA

LTKJava awaits the *ConnectionStatusEvent* indicator and throws an exception if the connection fails, as shown in Figure 6.13.

LTKJAVA

```
private void connect()
{
   // create client-initiated LLRP connection
   LLRPConnector connection = new LLRPConnector(this, READER_IP_ADDRESS);

   // connect to reader
```

```
   // LLRPConnector.connect waits for successful
   // READER_EVENT_NOTIFICATION from reader
   try
   {
      logger.info("Initiate LLRP connection to reader");
      ((LLRPConnector)connection).connect();
   }
   catch (LLRPConnectionAttemptFailedException e1)
   {
      e1.printStackTrace();
      System.exit(1);
   }
}
```

Figure 6.13 LTKJava – Creating/Validating Connection

6.4 Enabling the Impinj Extensions

To ensure compatibility with standard LLRP implementations, Octane Impinj extensions must be enabled each time the application connects (or re-connects) to the Reader.

Enabling Octane Impinj extensions requires creating an LLRP CUSTOM_MESSAGE that contains the IMPINJ_ENABLE_EXTENSIONS command. This is identical to the method you would use to create other Impinj custom message types. See the examples below for details.

6.4.1 LTKCPP

In this example, a **CIMPINJ_ENABLE_EXTENSIONS** object is created. Its *messageID* is set via the ***setMessageID()*** accessor***.*** An LLRP transaction is called and the response is validated. Both messages are freed before completing the operation.

LTKCPP

```
{
   CIMPINJ_ENABLE_EXTENSIONS * pCmd;
   CMessage * pRspMsg;
   CIMPINJ_ENABLE_EXTENSIONS_RESPONSE *pRsp;

   /*
   * Compose the command message
```

```
*/
pCmd = new CIMPINJ_ENABLE_EXTENSIONS();
pCmd->setMessageID(1);

/*
* Send the message, expect a certain type of response
*/
pRspMsg = transact(pCmd);

/*
* Done with the command message
*/
delete pCmd;

/*
* transact() returns NULL if something went wrong.
*/
if(NULL == pRspMsg)
{
   /* transact already tattled */
   return -1;
}

/*
* Cast to a CIMPINJ_ENABLE_EXTENSIONS_RESPONSE message.
*/
pRsp = (CIMPINJ_ENABLE_EXTENSIONS_RESPONSE *) pRspMsg;

/*
* Check the LLRPStatus parameter.
*/
if (0 != checkLLRPStatus(pRsp->getLLRPStatus(),
   "enableImpinjExtensions"))
{
   /* checkLLRPStatus already tattled */
   delete pRspMsg;
   return -1;
}

/*
```

```
    * Done with the response message.
    */
    delete pRspMsg;

    /*
    * Victory.
    */
    return 0;
}
```

Figure 6.14 LTKCPP – Enabling Impinj Extensions

6.4.2 LTKNET

In this example, a **MSG_IMPINJ_ENABLE_EXTENSIONS** object is created. Its *messageID* is set via the **MSG_ID** property**.** After it is created, the **LLRPClient**.***CUSTOM_MESSAGE*** is called. If a response is received, an explict cast will be required to convert the custom response back to the MSG_IMPINJ_ENABLE_EXTENSIONS_RESPONSE.

LTKNET

```
#region EnableExtensions
{
    Console.WriteLine("Enabling Impinj Extensions\\n");
    MSG_IMPINJ_ENABLE_EXTENSIONS
            imp_msg = new MSG_IMPINJ_ENABLE_EXTENSIONS();
    MSG_ERROR_MESSAGE msg_err;
    // imp_msg.MSG_ID doesn't need to be set as the library will default
    imp_msg.MSG_ID = 1;
    // Send the custom message and wait for 8 seconds
    MSG_CUSTOM_MESSAGE cust_rsp =
        reader.CUSTOM_MESSAGE(imp_msg, out msg_err, 8000);
    MSG_IMPINJ_ENABLE_EXTENSIONS_RESPONSE msg_rsp =
        cust_rsp as MSG_IMPINJ_ENABLE_EXTENSIONS_RESPONSE;

    if (msg_rsp != null)
    {
        if (msg_rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
        {
            Console.WriteLine(msg_rsp.LLRPStatus.StatusCode.ToString());
```

```
         reader.Close();
         return;
      }
   }
   else if (msg_err != null)
   {
      Console.WriteLine(msg_err.ToString());
      reader.Close();
      return;
   }
   else
   {
      Console.WriteLine(
         "Enable Extensions Command Timed out\\n");
      reader.Close();
      return;
   }
}
```

Figure 6.15 LTKNET – Enabling Impinj Extensions

6.4.3 LTKJAVA

In this example a MSG_IMPINJ_ENABLE_EXTENSIONS object is created. Set the *messageID* via the ***setMessageID*** methodAfter it is created, the transaction is performed. If it receives a response, the status code is verified for a successful response.

LTKJAVA

```
private void enableImpinjExtensions()
{
   LLRPMessage response;
   try
   {
      logger.info("IMPINJ_ENABLE_EXTENSIONS ...");
      IMPINJ_ENABLE_EXTENSIONS ena =
         new IMPINJ_ENABLE_EXTENSIONS();
      ena.setMessageID(getUniqueMessageID());
      response = connection.transact(ena, 10000);
      StatusCode status =
```

```
        ((IMPINJ_ENABLE_EXTENSIONS_RESPONSE)response)
          .getLLRPStatus().getStatusCode();

      if (status.equals(new StatusCode("M_Success")))
      {
        logger.info(
          "IMPINJ_ENABLE_EXTENSIONS was successful");
      }
      else
      {
        logger.info(response.toXMLString());
        logger.info(
          "IMPINJ_ENABLE_EXTENSIONS Failure");
        System.exit(1);
      }
    }
    catch (InvalidLLRPMessageException ex)
    {
      logger.error(
        "Could not process IMPINJ_ENABLE_EXTENSIONS response");
      System.exit(1);
    }
    catch (TimeoutException ex)
    {
      logger.error(
        "Timeout Waiting for IMPINJ_ENABLE_EXTENSIONS response");
      System.exit(1);
    }
}
```

Figure 6.16 LTKJAVA – Enabling Impinj Extensions

6.5 Debugging with LTK

All Windows-compatible LTK libraries contain accompanying debug versions that are suitable for common Visual Studio debugging. All Linux-compatible LTK libraries provide integrated symbol debugging information. Debug information can be stripped using ‘strip’ to save program space.

When you work with LLRP messages, consider keeping a reference printout of the LTK-XML message version for a debug log file or console file for examination. The following code samples

show how to print LTK-XML from internal LTK message objects. Note: these code samples come from the program sample, and are not logging to a file, but simply to the output device common for the related platform.

6.5.1 LTKC

In LTKC, a built-in function***, LLRP_toXMLString(),*** prints the LLRP or Impinj object in LTK-XML format. The function takes any CELEMENT data structure including messages and parameters.

LTKC

```
void printXMLMessage (LLRP_tSMessage * pMessage)
{
   char aBuf[100*1024];

   /*
   * Convert the message to an XML string.
   * This fills the buffer with either the XML string
   * or an error message. The return value could
   * be checked.
   */
   LLRP_toXMLString(&pMessage->elementHdr, aBuf, sizeof aBuf);

   /* Print the LTK-XML Text to the standard output.*/
   printf("%s", aBuf);
}
```

Figure 6.17 LTKC – Printing LLRP Messages

6.5.2 LTKCPP

In LTKCPP, the built-in member function for all messages and parameters, ***toXMLString***, prints the LLRP or Impinj object in LTK-XML format.

LTKCPP

```
void CMyApplication::printXMLMessage (CMessage * pMessage)
{
   char aBuf[100*1024];
```

```
    /*
    * Convert the message to an XML string.
    * This fills the buffer with either the XML string
    * or an error message. The return value could
    * be checked.
    */

    pMessage->toXMLString(aBuf, sizeof aBuf);

    /* Print the LTK-XML Text to the standard output. */
    printf("%s", aBuf);
}
```

Figure 6.18 LTKCPP – Printing LLRP Messages

6.5.3 LTKNET

In LTKNET, the ***ToString()*** method, available for all LLRP messages and parameters, serializes the object as LTK-XML.

LTKNET

```
else if (msg_err != null)
{
    Console.WriteLine(msg_err.ToString());
    reader.Close();
    return;
}
```

Figure 6.19 LTKNET – Printing LLRP Messages

6.5.4 LTKJava

In LTKJAVA, the ***toXMLString()*** method, available for all LLRP messages and parameters, serializes the object as LTK-XML. In addition, LTKJAVA has a log4j mechanism that can enable detailed logs. When trying to understand your LTKJAVA program, it is helpful to set the log level for the LTK.

LTKJAVA

```
Logger.getRootLogger().setLevel(Level.DEBUG);
if (status.equals(new StatusCode("M_Success")))
{
   logger.info("IMPINJ_ENABLE_EXTENSIONS was successful");
}
else
{
   logger.info(response.toXMLString());
   logger.info("IMPINJ_ENABLE_EXTENSIONS Failure");
   System.exit(1);

}
```

Figure 6.20 LTKJava – Printing LLRP Messages

7 Creating a Simple Impinj LTK Application

The most basic LLRP application ignores the Reader capabilities (because it has prior knowledge of everything that is relevant), and uses all of the Reader's default values. It also adds the simplest *ROSpec,* controlling the Reader manually from the application. The Impinj LTK distributions include this example, **'docsample1'.**

7.1 Application Flow

The following steps initiate and send an LLRP message to the Reader.

1. Initialize the Library.
2. Connect to the Reader.
3. Enable Impinj Extensions.
4. Use the factory default LLRP configuration to ensure that the Reader is in a known state (since we are relying on the default Reader configuration for this simple example).
5. **ADD_ROSPEC** tells the Reader to perform an inventory.
6. **ENABLE_ROSPEC**
7. **START_ROSPEC** starts the inventory operation.
8. Process RFID Data (EPC, RSSI, Timestamp).

Sections 6.2, 6.3, and 6.4 in this document provide details for Steps 1, 2, and 3.

Steps 3-7 involve sending LLRP messages to the Reader. In terms of LTK-XML, the following message is exchanged between the application and the Reader during a successful transaction.

LTK-XML

```
<!-- Enabled Impinj Extensions -->
<Impinj:IMPINJ_ENABLE_EXTENSIONS MessageID="1">
</Impinj:IMPINJ_ENABLE_EXTENSIONS >

<!-Readers expected response -->
<Impinj:IMPINJ_ENABLE_EXTENSIONS_RESPONSE MessageID="1">
   <llrp:LLRPStatus>
      <StatusCode>M_Success</StatusCode>
      <ErrorDescription></ErrorDescription>
   </llrp:LLRPStatus>
</Impinj:IMPINJ_ENABLE_EXTENSIONS_RESPONSE>
```

```
<!- Use Reader Factory Default settings -->
<SET_READER_CONFIG MessageID="2">
   <ResetToFactoryDefault>true</ResetToFactoryDefault>

</SET_READER_CONFIG>

<!-Readers expected response -->
<SET_READER_CONFIG_RESPONSE MessageID="2">
   <LLRPStatus>
      <StatusCode>M_Success</StatusCode>
      <ErrorDescription></ErrorDescription>
   </LLRPStatus>

</SET_READER_CONFIG_RESPONSE>

<!-- Add a basic Gen2 ROSpec -->
<ADD_ROSPEC MessageID="3">
   <ROSpec> [[See below for details]]</ROSpec>
</ADD_ROSPEC>

<!-Readers expected response -->
<ADD_ROSPEC_RESPONSE MessageID="3">
   <LLRPStatus>
      <StatusCode>M_Success</StatusCode>
      <ErrorDescription></ErrorDescription>
   </LLRPStatus>
</ADD_ROSPEC_RESPONSE>

<!-Enable the ROSpec -->
<ENABLE_ROSPEC MessageID="4">
   <ROSpecID>1111</ROSpecID>\ <!-Make sure this matches the ID
   for the ROSpec above -->
</ENABLE_ROSPEC>

<!-Readers expected response -->
<ENABLE_ROSPEC_RESPONSE MessageID="4">
   <LLRPStatus>
      <StatusCode>M_Success</StatusCode>
      <ErrorDescription></ErrorDescription>
   </LLRPStatus>
```

```
</ENABLE_ROSPEC_RESPONSE>

<!-Start the ROSpec -->
<START_ROSPEC MessageID="5">
   <ROSpecID>1111</ROSpecID>\ <!-Make sure this matches the ID
   for the ROSpec above -->
</START_ROSPEC>

<!-Readers expected response -->
<START_ROSPEC_RESPONSE MessageID="5">
   <LLRPStatus>
      <StatusCode>M_Success</StatusCode>
      <ErrorDescription></ErrorDescription>
   </LLRPStatus>

</START_ROSPEC_RESPONSE>
```

Figure 7.1 LTK-XML – Basic Application Sequence

7.2 Setup

Section 4.4 in this document provides detailed setup information. It is assumed that the programmer has installed:

- Tools, and has compiled the application with support for the LTK, as described in Section 6.1.
- Library is initialized as described in Section 6.2, and the Reader connection is established.
- Impinj Octane extensions are enabled, as described in Section 6.4.

7.3 Factory Defaults for the Reader

Readers come with predictable LLRP default values that have uses in a wide variety of RFID applications. For this simple example, we assume that the LLRP default values are sufficient for the use case. The LLRP SET_READER_CONFIG message has a single Boolean field to restore the Reader's LLRP settings to their default values. Note, the LLRP factory default field does not change other properties on the Reader, and only affects the LLRP configuration, *ROSpecs*, and *AccessSpecs*.

The examples in the following sections show how this is performed.

7.3.1 LTKCPP

A **CSET_READER_CONFIG** object is created. The related *messageID* is set by the ***setMesageID()*** accessor, and the *FactoryDefault* field is set by the ***setResetToFactoryDefault()*** accessor. An LLRP transaction is called and the response validates. Both messages are freed before completing the operation.

LTKCPP

```
{
   CSET_READER_CONFIG * pCmd;
   CMessage * pRspMsg;
   CSET_READER_CONFIG_RESPONSE *pRsp;

   /*
   * Compose the command message
   */
   pCmd = new CSET_READER_CONFIG();
   pCmd->setMessageID(2);
   pCmd->setResetToFactoryDefault(1);

   /*
   * Send the message, expect a certain type of response
   */

   pRspMsg = transact(pCmd);
   /*
   * Done with the command message
   */
   delete pCmd;

   /*
   * transact() returns NULL if something went wrong.
   */

   if(NULL == pRspMsg)
   {
   /* transact already tattled */
   return -1;
   }

   /*
```

```
   * Cast to a SET_READER_CONFIG_RESPONSE message.
   */
   pRsp = (CSET_READER_CONFIG_RESPONSE *) pRspMsg;

   /*
   * Check the LLRPStatus parameter.
   */
   if(0 != checkLLRPStatus(pRsp->getLLRPStatus(),
   "resetConfigurationToFactoryDefaults"))
   {
      /* checkLLRPStatus already tattled */
      delete pRspMsg;
      return -1;
   }

   /*
   * Done with the response message.
   */
   delete pRspMsg;

   /*
   * Victory.
   */
   return 0;

}
```

Figure 7.2 LTKCPP – Factory Default LLRP Configuration

7.3.2 LTKNET

A MSG_SET_READER_CONFIG object is created. The *messageID* is set by the MSG_ID property, and the ***FactoryDefault*** field is set by the **ResetToFactoryDefault** property. The **LLRPClient**.***SET_READER_CONFIG*** transaction is called and the response validates.

LTKNET

```
#region FactoryDefault
{
   Console.WriteLine("Factory Default the Reader\\n");
```

```
    // factory default for the reader
    MSG_SET_READER_CONFIG msg_cfg =
          new MSG_SET_READER_CONFIG();

    MSG_ERROR_MESSAGE msg_err;
    msg_cfg.ResetToFactoryDefault = true;
    msg_cfg.MSG_ID = 2; //Doesn't need to be set. The library will default

    // If SET_READER_CONFIG affects antennas it could take several
    // seconds to return
    MSG_SET_READER_CONFIG_RESPONSE rsp_cfg =
       reader.SET_READER_CONFIG(msg_cfg, out msg_err, 12000);

    if (rsp_cfg != null)
    {
       if (rsp_cfg.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
       {
          Console.WriteLine(rsp_cfg.LLRPStatus.StatusCode.ToString());
          reader.Close();
          return;
       }
    }

    else if (msg_err != null)
    {
       Console.WriteLine(msg_err.ToString());
       reader.Close();
       return;
    }

    else
    {
       Console.WriteLine("SET_READER_CONFIG Command Timed out\\n");
       reader.Close();
       return;
    }
}

#endregion
```

Figure 7.3 LTKNET – Factory Default LLRP Configuration

7.3.3 LTKJAVA

A MSG_SET_READER_CONFIG object is created. The *messageID* is set by the **setMessageID** method, and the *FactoryDefault* field is set by the **setResetToFactoryDefault** method. The transact method is called and the response validates.

LTKJAVA

```
private void factoryDefault()
{
   LLRPMessage response;
   try
   {
      // factory default the reader
      logger.info("SET_READER_CONFIG with factory default ...");

      SET_READER_CONFIG set = new SET_READER_CONFIG();
      set.setMessageID(getUniqueMessageID());
      set.setResetToFactoryDefault(new Bit(true));
      response = connection.transact(set, 10000);

      // check whether ROSpec addition was successful
      StatusCode status =
         ((SET_READER_CONFIG_RESPONSE)response).
            getLLRPStatus().getStatusCode();

      if (status.equals(new StatusCode("M_Success")))
      {
         logger.info("SET_READER_CONFIG Factory Default was successful");
      }
      else
      {
         logger.info(response.toXMLString());
         logger.info("SET_READER_CONFIG Factory Default Failure");
         System.exit(1);
      }
   }
   catch (Exception e)
   {
```

```
        e.printStackTrace();
        System.exit(1);
    }
}
```

Figure 7.4 LTKJava – Factory Default LLRP Configuration

7.4 Add an ROSpec

The *ROSpec* contains the Reader's instructions for inventory operation. In this simple example, we assume that the application manually controls the Reader start and stop, and uses all of the Reader's default values for reporting of tag data.

When adding a *ROSpec*, the *ROSpec* state must be set to **Disabled**. An error is returned by the Reader (not the LTK) when adding a *ROSpec* in the enabled or active state.

To allow only manual starting and stopping of the *ROSpec*, the *RoSpecStartTrigger* and *RoSpecStopTrigger* must be set to **Null**. The *RoSpecStopTrigger Duration* value is ignored when the trigger type is **Null**, but we recommend setting this field to zero.

LLRP *ROSpec* does not support *Priority* in Speedway or, which supports a single *ROSpec*. This field must be set to zero.

The simplest Antenna Inventory spec (*AISpec*) describes the desired Reader behavior. The Reader performs a Gen2 Inventory on all antennas that have no *AISpec* specific stop conditions. To specify all antennas, LLRP reserves the value of zero (0).

Both the *RoSpec* and the *InventoryParameterSpec* require an identification number, the *ROSpecID* and *InventoryParameterSpecID* respectively. The *RoSpecID* is used by the Reader when receiving ENABLE_ROSPEC, DISABLE_ROSPEC, START_ROSPEC, or STOP_ROSPEC messages, and when reporting the *ROSpecID* in the *TagReportData* parameters. The application assigns a number that is only used for identification purposes. We recommend using a number significant to your application. LTK and LLRP allow any 32-bit number (except 0) for these IDs.

The LTK-XML representation of this simple *ROSpec* is shown below:

LTK-XML

```
<?xml version="1.0"?>
<ADD_ROSPEC
   xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:llrp="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.6"
```

```
   xsi:schemaLocation="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0
   http://www.llrp.org/ltk/schema/core/encoding/xml/1.0/llrp.xsd
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6/impinj.xsd"

MessageID="0">
   <ROSpec>
      <ROSpecID>1111</ROSpecID>
      <Priority>0</Priority>
      <CurrentState>Disabled</CurrentState>
      <ROBoundarySpec>
         <ROSpecStartTrigger>
            <ROSpecStartTriggerType>Null</ROSpecStartTriggerType>
         </ROSpecStartTrigger>
         <ROSpecStopTrigger>
            <ROSpecStopTriggerType>Null</ROSpecStopTriggerType>
            <DurationTriggerValue>0</DurationTriggerValue>
         </ROSpecStopTrigger>
      </ROBoundarySpec>

      <AISpec>
         <AntennaIDs>0</AntennaIDs>
         <AISpecStopTrigger>
            <AISpecStopTriggerType>Null</AISpecStopTriggerType>
            <DurationTrigger>0</DurationTrigger>
         </AISpecStopTrigger>
         <InventoryParameterSpec>
            <InventoryParameterSpecID>1234</InventoryParameterSpecID>
            <ProtocolID>EPCGlobalClass1Gen2</ProtocolID>
         </InventoryParameterSpec>
      </AISpec>
   </ROSpec>

</ADD_ROSPEC>
```

Figure 7.5 LTK-XML – Simple ROSpec

7.4.1 LTKCPP

In LTKCPP, each of the parameters within the *ROSpec* needs to be created individually. The individual fields within the parameter also need to be explicitly set. After this has been completed, the parameter is inserted into its parent parameter or message. As an example. shown in Figure 7.6, the **CROSpecStartTrigger** and **CROSpecStopTrigger** parameters are created and then inserted into the newly created **CROBoundarySpec.** Repeat this procedure for each of the parameters within the *ROSpec*.

Of particular note here is the use of "lists" or vectors of elements. The *AISpec* provides a list of active antennas. In this case, we are setting the antenna set to '0', or all antennas. To do so, we need to create the ***llrp_u16v_t(1)*** list and assign its first element to the value '0'.

Notice that, although each object is constructed individually, all of the objects are freed by a single ***delete pCmd;*** statement. The delete operator automatically frees all message sub-components that were allocated in the previous **new** operations.

The ability to construct *ROSpecs* (or other messages or parameters) from XML is not available in the LTKCPP.

LTKCPP

```
CROSpecStartTrigger * pROSpecStartTrigger =
   new CROSpecStartTrigger();
pROSpecStartTrigger->setROSpecStartTriggerType(
   ROSpecStartTriggerType_Null);
CROSpecStopTrigger * pROSpecStopTrigger = new CROSpecStopTrigger();
pROSpecStopTrigger->setROSpecStopTriggerType(ROSpecStopTriggerType_Null);
pROSpecStopTrigger->setDurationTriggerValue(0); /* n/a */
CROBoundarySpec * pROBoundarySpec = new CROBoundarySpec();
pROBoundarySpec->setROSpecStartTrigger(pROSpecStartTrigger);
pROBoundarySpec->setROSpecStopTrigger(pROSpecStopTrigger);
CAISpecStopTrigger * pAISpecStopTrigger = new CAISpecStopTrigger();
pAISpecStopTrigger->setAISpecStopTriggerType(
   AISpecStopTriggerType_Null);
pAISpecStopTrigger->setDurationTrigger(0);
CInventoryParameterSpec * pInventoryParameterSpec =
   new CInventoryParameterSpec();
pInventoryParameterSpec->setInventoryParameterSpecID(1234);
pInventoryParameterSpec->setProtocolID(AirProtocols_EPCGlobalClass1Gen2);
llrp_u16v_t AntennaIDs = llrp_u16v_t(1);
AntennaIDs.m_pValue[0] = 0; /* All */
CAISpec * pAISpec = new CAISpec();
```

```
pAISpec->setAntennaIDs(AntennaIDs);
pAISpec->setAISpecStopTrigger(pAISpecStopTrigger);
pAISpec->addInventoryParameterSpec(pInventoryParameterSpec);
CROSpec * pROSpec = new CROSpec();
pROSpec->setROSpecID(1111);
pROSpec->setPriority(0);
pROSpec->setCurrentState(ROSpecState_Disabled);
pROSpec->setROBoundarySpec(pROBoundarySpec);
pROSpec->addSpecParameter(pAISpec);
CADD_ROSPEC * pCmd;
CMessage * pRspMsg;
CADD_ROSPEC_RESPONSE * pRsp;

/*
* Compose the command message.
* N.B.: After the message is composed, all the parameters
* constructed, immediately above, are considered "owned"
* by the command message. When it is destructed so
* too will the parameters be.
*/
pCmd = new CADD_ROSPEC();
pCmd->setMessageID(3);
pCmd->setROSpec(pROSpec);

/*  Send the message, expect a certain type of response */
pRspMsg = transact(pCmd);

/*
* Done with the command message.
* N.B.: And the parameters
*/
delete pCmd;

/* transact() returns NULL if something went wrong. */
if(NULL == pRspMsg)
{
   /* transact already tattled */
   return -1;
}
```

```
/* Cast to a ADD_ROSPEC_RESPONSE message. */
pRsp = (CADD_ROSPEC_RESPONSE *) pRspMsg;

/* Check the LLRPStatus parameter. */
if(0 != checkLLRPStatus(pRsp->getLLRPStatus(), "addROSpec"))
{
   /* checkLLRPStatus already tattled */
   delete pRspMsg;
   return -1;
}

/* Done with the response message. */
delete pRspMsg;

/* Victory. */
return 0;
```

Figure 7.6 LTKCPP – Constructing a Simple ROSpec

7.4.2 LTKNET

There are two ways to build *ROSpecs* and other LLRP messages and parameters in LTKNET:

- Programmatically with LTKNET C# objects
- By importing LTK-XML descriptions

For simple messages, such as those as described in Section 7.3, it is easier and faster to construct the LLRP objects programmatically. However, for large or complex messages such as ADD_ROSPEC, ADD_ACCESSSPEC, and SET_READER_CONFIG, there are times where XML-based construction is better.

Generally, for use cases where the configuration is relatively static, but where you want to “tune” the Reader performance in the field, reading ADD_ROSPEC and ADD_ACCESSSPEC messages from a file offers the greatest flexibility. However, if the user of the application requires fine-grained control over specific LLRP options via a GUI or other mechanism, it might be easier using the programmatic approach.

The object-based construction is shown in Figure 7.7. For an example of the LTK-XML based construction, see Section 8.5.1.

LTKNET

```
#region ADDRoSpecWithObjects
{
   Console.WriteLine("Adding RoSpec\\n");

   // set up the basic parameters in the ROSpec. Use all the defaults
   // from the reader
   MSG_ADD_ROSPEC msg = new MSG_ADD_ROSPEC();
   MSG_ERROR_MESSAGE msg_err;
   msg.ROSpec = new PARAM_ROSpec();
   msg.ROSpec.CurrentState = ENUM_ROSpecState.Disabled;
   msg.ROSpec.Priority = 0x00;
   msg.ROSpec.ROSpecID = 1111;

   // setup the start and stop triggers in the Boundary Spec
   msg.ROSpec.ROBoundarySpec = new PARAM_ROBoundarySpec();
   msg.ROSpec.ROBoundarySpec.ROSpecStartTrigger = new
          PARAM_ROSpecStartTrigger();
   msg.ROSpec.ROBoundarySpec.ROSpecStartTrigger.ROSpecStartTriggerType =
          ENUM_ROSpecStartTriggerType.Null;
   msg.ROSpec.ROBoundarySpec.ROSpecStopTrigger = new PARAM_ROSpecStopTrigger();
   msg.ROSpec.ROBoundarySpec.ROSpecStopTrigger.ROSpecStopTriggerType =
          ENUM_ROSpecStopTriggerType.Null;
   msg.ROSpec.ROBoundarySpec.ROSpecStopTrigger.DurationTriggerValue = 0;

   // ignored by reader
   // Add a single Antenna Inventory to the ROSpec
   msg.ROSpec.SpecParameter = new UNION_SpecParameter();
   PARAM_AISpec aiSpec = new PARAM_AISpec();
   aiSpec.AntennaIDs = new UInt16Array();
   aiSpec.AntennaIDs.Add(0); // all antennas
   aiSpec.AISpecStopTrigger = new PARAM_AISpecStopTrigger();
   aiSpec.AISpecStopTrigger.AISpecStopTriggerType =
         ENUM_AISpecStopTriggerType.Null;

   // use all the defaults from the reader. Just specify the minimum required
   aiSpec.InventoryParameterSpec = new PARAM_InventoryParameterSpec[1];
   aiSpec.InventoryParameterSpec[0] = new PARAM_InventoryParameterSpec();
   aiSpec.InventoryParameterSpec[0].InventoryParameterSpecID = 1234;
   aiSpec.InventoryParameterSpec[0].ProtocolID =
          ENUM_AirProtocols.EPCGlobalClass1Gen2;
```

```
    msg.ROSpec.SpecParameter.Add(aiSpec);
    MSG_ADD_ROSPEC_RESPONSE rsp =
       reader.ADD_ROSPEC(msg, out msg_err, 12000);

    if (rsp != null)
    {
       if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
       {
          Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
          reader.Close();
          return;
       }
    }
    else if (msg_err != null)
    {
       Console.WriteLine(msg_err.ToString());
       reader.Close();
       return;
    }
    else
    {
       Console.WriteLine(
          "ADD_ROSPEC Command Timed out\\n");
      reader.Close();
       return;
    }
}
#endregion
```

Figure 7.7 LTKNET – Constructing a Simple ROSpec with Objects

7.4.3 LTKJava

In LTKJAVA, there are two ways to build *ROSpecs* (as well as other LLRP messages and parameters):

- Programmatically with LTKJAVA objects
- By importing LTK-XML descriptions

For simple messages, such as those described in Section 7.3, it is easier and faster to construct the LLRP objects programmatically. However, for large or complex messages such as ADD_ROSPEC,

ADD_ACCESSSPEC, and SET_READER_CONFIG, there are times where XML-based construction is a better choice.

Generally, for use cases where the configuration is relatively static, but where you want to "tune" the performance of the Reader in the field, reading ADD_ROSPEC and ADD_ACCESSSPEC messages from a file offers the most flexibility. If the user of the application requires fine-grained control over specific LLRP options via a GUI or other mechanism, the programmatic approach might be easier to use.

The object-based construction is shown in Figure 7.8. For an example of the LTK-XML based construction, see Section 8.5.1.

LTKJAVA

```
private ADD_ROSPEC buildROSpecFromObjects()
{
   logger.info("Building ADD_ROSPEC message from scratch ...");
   ADD_ROSPEC addRoSpec = new ADD_ROSPEC();
   addRoSpec.setMessageID(getUniqueMessageID());
   rospec = new ROSpec();

   // set up the basic info for the RO Spec.
   rospec.setCurrentState(new ROSpecState(ROSpecState.Disabled));
   rospec.setPriority(new UnsignedByte(0));
   rospec.setROSpecID(new UnsignedInteger(12345));

   // set the start and stop conditions for the ROSpec.
   // For now, we will start and stop manually
   ROBoundarySpec boundary = new ROBoundarySpec();
   ROSpecStartTrigger start = new ROSpecStartTrigger();
   ROSpecStopTrigger stop = new ROSpecStopTrigger();
   start.setROSpecStartTriggerType(new
      ROSpecStartTriggerType(ROSpecStartTriggerType.Null));
   stop.setROSpecStopTriggerType(new
      ROSpecStopTriggerType(ROSpecStopTriggerType.Null));
   stop.setDurationTriggerValue(new UnsignedInteger(0));
   boundary.setROSpecStartTrigger(start);
   boundary.setROSpecStopTrigger(stop);
   rospec.setROBoundarySpec(boundary);

   // Set up when we want to see the report
   report.setROReportTrigger(new ROReportTriggerType(ROReportTriggerType.
      Upon_N_Tags_Or_End_Of_ROSpec));
```

```
    report.setN(new UnsignedShort(1));
    rospec.setROReportSpec(report);

    // set up what we want to do in the ROSpec. In this case
    // build the simples inventory on all channels using defaults
    AISpec aispec = new AISpec();

    // what antennas to use.
    UnsignedShortArray ants = new UnsignedShortArray();
    ants.add(new UnsignedShort(0)); // 0 means all antennas
    aispec.setAntennaIDs(ants);

    // set up the AISpec stop condition and options for inventory
    AISpecStopTrigger aistop = new AISpecStopTrigger();
    aistop.setAISpecStopTriggerType(new
    AISpecStopTriggerType(AISpecStopTriggerType.Null));
    aistop.setDurationTrigger(new UnsignedInteger(0));
    aispec.setAISpecStopTrigger(aistop);

    // set up any override configuration. none in this case
    InventoryParameterSpec ispec = new InventoryParameterSpec();
    ispec.setAntennaConfigurationList(null);
    ispec.setInventoryParameterSpecID(new UnsignedShort(23));
    ispec.setProtocolID(new AirProtocols(AirProtocols.EPCGlobalClass1Gen2));

    List<InventoryParameterSpec> ilist = new ArrayList<InventoryParameterSpec>();
    ilist.add(ispec);
    aispec.setInventoryParameterSpecList(ilist);

    List<SpecParameter> slist = new ArrayList<SpecParameter>();
    slist.add(aispec);
    rospec.setSpecParameterList(slist);
    addRoSpec.setROSpec(rospec);
    return addRoSpec;
}
```

Figure 7.8 LTKJava – Constructing a Simple ROSpec with Objects

7.5 Enable an ROSpec

An *ROSpec* must be added in the Disabled state, however the *ROSpec* must be enabled before it responds to a START_ROSPEC message. The examples below follow the same pattern as the factory default setup described in Section 7.3. Note, when enabling and starting *ROSpecs*, it is important to use the *ROSpecID* that matches the ID from the *ROSpec* you added in Section 7.4.

LTKCPP

```
{
   CENABLE_ROSPEC * pCmd;
   CMessage * pRspMsg;
   CENABLE_ROSPEC_RESPONSE * pRsp;

   /* Compose the command message */
   pCmd = new CENABLE_ROSPEC();
   pCmd->setMessageID(4);
   pCmd->setROSpecID(1111);

   /* Send the message, expect a certain type of response */
   pRspMsg = transact(pCmd);

   /* Done with the command message */
   delete pCmd;

   /* transact() returns NULL if something went wrong. */
   if(NULL == pRspMsg)
   {
      /* transact already tattled */
      return -1;
   }

   /*  Cast to a ENABLE_ROSPEC_RESPONSE message. */
   pRsp = (CENABLE_ROSPEC_RESPONSE *) pRspMsg;

   /* Check the LLRPStatus parameter. */
   if(0 != checkLLRPStatus(pRsp->getLLRPStatus(), "enableROSpec"))
   {
      /* checkLLRPStatus already tattled */
      delete pRspMsg;
      return -1;
   }
```

```
    /* Done with the response message. */
    delete pRspMsg;

    /* Victory. */
    return 0;
}
```

Figure 7.9 LTKCPP – Enable an ROSpec

LTKNET

```
#region EnableRoSpec
{
    Console.WriteLine("Enabling RoSpec\\n");
    MSG_ENABLE_ROSPEC msg = new MSG_ENABLE_ROSPEC();
    MSG_ERROR_MESSAGE msg_err;
    msg.ROSpecID = 1111; // this better match the ROSpec we created above
    MSG_ENABLE_ROSPEC_RESPONSE rsp =
        reader.ENABLE_ROSPEC(msg, out msg_err, 12000);

    if (rsp != null)
    {
        if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
        {
            Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
            reader.Close();
            return;
        }
    }
    else if (msg_err != null)
    {
        Console.WriteLine(msg_err.ToString());
        reader.Close();
        return;
    }
    else
    {
        Console.WriteLine("ENABLE_ROSPEC Command Timed out\\n");
        reader.Close();
        return;
```

```
    }
}
#endregion
```

Figure 7.10 LTKNET – Enable an ROSpec

LTKJAVA

```
private void enable() {
   LLRPMessage response;
   try
   {
      // factory default the reader
      logger.info("ENABLE_ROSPEC ...");
      ENABLE_ROSPEC ena = new ENABLE_ROSPEC();
      ena.setMessageID(getUniqueMessageID());
      ena.setROSpecID(rospec.getROSpecID());
      response = connection.transact(ena, 10000);

      // check whether ROSpec addition was successful
      StatusCode status =
         ((ENABLE_ROSPEC_RESPONSE)response).getLLRPStatus().
             getStatusCode();

     if (status.equals(new StatusCode("M_Success")))
     {
        logger.info("ENABLE_ROSPEC was successful");
     }
     else
     {
        logger.error(response.toXMLString());
        logger.info("ENABLE_ROSPEC_RESPONSE failed ");
        System.exit(1);
     }
  }
  catch (Exception e)
  {
     e.printStackTrace();
     System.exit(1);
  }
}
```

Figure 7.11 LTKJAVA – Enable an ROSpec

7.6 Start an ROSpec

In this example, we configured the *ROSpec* without start or stop triggers. Consequently the application must start and stop the Reader by using the START_ROSPEC and STOP_ROSPEC messages respectively. In applications that use internal Reader triggering, this step is not necessary. These examples follow the identical pattern as the factory default described in Section 7.3. Note, when enabling and starting *ROSpecs*, it is important to use the *ROSpecID* that matches the ID from the *ROSpec* that you added in Section 7.4. The examples below start the *ROSpec* in LTKCPP, LTKNET, and LTKJAVA formats.

LTKCPP

```
{
   CSTART_ROSPEC * pCmd;
   CMessage * pRspMsg;
   CSTART_ROSPEC_RESPONSE * pRsp;

   /* Compose the command message */
   pCmd = new CSTART_ROSPEC();
   pCmd->setMessageID(5);
   pCmd->setROSpecID(1111);

   /*  Send the message, expect a certain type of response */
   pRspMsg = transact(pCmd);

   /* Done with the command message */
   delete pCmd;

   /* transact() returns NULL if something went wrong. */
   if(NULL == pRspMsg)
   {
      /* transact already tattled */
      return -1;
   }

   /* Cast to a START_ROSPEC_RESPONSE message. */
   pRsp = (CSTART_ROSPEC_RESPONSE *) pRspMsg;

   /* Check the LLRPStatus parameter. */
```

```
    if(0 != checkLLRPStatus(pRsp->getLLRPStatus(), "startROSpec"))
    {
        /* checkLLRPStatus already tattled */
        delete pRspMsg;
        return -1;
    }
    /* Done with the response message. */
    delete pRspMsg;

    /* Victory. */
    return 0;
}
```

Figure 7.12 LTKCPP – Start an ROSpec

LTKNET

```
#region StartRoSpec
{
    Console.WriteLine("Starting RoSpec\\n");
    MSG_START_ROSPEC msg = new MSG_START_ROSPEC();
    MSG_ERROR_MESSAGE msg_err;
    msg.ROSpecID = 1111; // this better match the RoSpec we created above
    MSG_START_ROSPEC_RESPONSE rsp =
        reader.START_ROSPEC(msg, out msg_err, 12000);

    if (rsp != null)
    {
        if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
        {
            Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
            reader.Close();
            return;
        }
    }
    else if (msg_err != null)
    {
        Console.WriteLine(msg_err.ToString());
        reader.Close();
        return;
    }
```

```
  else
  {
     Console.WriteLine("START_ROSPEC Command Timed out\\n");
     reader.Close();
     return;
  }
}
#endregion
```

Figure 7.13 LTKNET – Start an ROSpec

LTKJAVA

```
private void start()
{
   LLRPMessage response;
   try
   {
      logger.info("START_ROSPEC ...");
      START_ROSPEC start = new START_ROSPEC();
      start.setMessageID(getUniqueMessageID());
      start.setROSpecID(rospec.getROSpecID());
      response = connection.transact(start, 10000);
      // check whether ROSpec addition was successful
      StatusCode status =
         ((START_ROSPEC_RESPONSE)response).getLLRPStatus().
               getStatusCode();
      if (status.equals(new StatusCode("M_Success")))
      {
         logger.info("START_ROSPEC was successful");
      }
      else
      {
         logger.error(response.toXMLString());
         logger.info("START_ROSPEC_RESPONSE failed ");
         System.exit(1);
      }
   }
   catch (Exception e)
   {
      e.printStackTrace();
```

```
      System.exit(1);
   }
}
```

Figure 7.14 LTKJava – Start an ROSpec

7.7 Handling Tag Report Data

Data is returned to the Reader via LLRP RO_ACCCESS_REPORT messages. The handling of these messages is notably different in the different LTK implementations in the examples below.

7.7.1 LTKCPP

In LTKCPP, a ***recvMessage(int timeout)*** method allows the caller to block received messages, with a timeout specified in milliseconds. Upon exiting from this call, the application needs to determine whether a message was received. If a message was received, the application examines the message type and dispatches the message accordingly. Dispatching the message requires that the application tests the message type. Each type of message or parameter contains a static type descriptor that can be compared against the descriptor within an instance of a message or parameter, as described in Section 5.1.5. The object can then be cast back to that type for handling. The example in Figure 7.15 is shown in LTKCPP:

LTKCPP

```
startTime = time(NULL);
pMessage = recvMessage(1000);

/* validate the timestamp */
tempTime = time(NULL);
if(difftime(tempTime, startTime) > 1000)
{
   bDone=1;
}
if(NULL == pMessage)
{
   continue;
}

/*
```

```
* What happens depends on what kind of message
* received. Use the type label (m_pType) to
* discriminate message types.
*/
pType = pMessage->m_pType;

/* Is it a tag report? If so, print it out. */
if(&CRO_ACCESS_REPORT::s_typeDescriptor == pType)
{
   CRO_ACCESS_REPORT * pNtf;
   pNtf = (CRO_ACCESS_REPORT *) pMessage;
   printTagReportData(pNtf);
}

/*
* Is it a reader event? This example only recognizes
* AntennaEvents.
*/
else if(&CREADER_EVENT_NOTIFICATION::s_typeDescriptor == pType)
{
   CREADER_EVENT_NOTIFICATION *pNtf;
   CReaderEventNotificationData *pNtfData;
   pNtf = (CREADER_EVENT_NOTIFICATION *) pMessage;
   pNtfData = pNtf->getReaderEventNotificationData();
   if(NULL != pNtfData)
   {
      handleReaderEventNotification(pNtfData);
   }
   else

   {
      printf("WARNING: READER_EVENT_NOTIFICATION without data\\n");
   }
```

Figure 7.15 LTKCPP –Receiving Asynchronous Reader Messages

Tag reports are returned in a list. To access an individual report, the list can be iterated with a standard STL iterator as shown in Figure 7.16.

```
std::list<CTagReportData *>::iterator Cur;
unsigned int nEntry = 0;

/* Loop through again and print each entry. */
for(
   Cur = pRO_ACCESS_REPORT->beginTagReportData();
   Cur != pRO_ACCESS_REPORT->endTagReportData();
   Cur++)
{
   printOneTagReportData(*Cur);
}
```

Figure 7.16 LTKCPP – Extracting Individual Tag Reports

7.7.2 LTKNET

In LTKNET, the **LLRPClient** object contains three event delegates to handle the three different types of messages initiated by the Reader:

- RO_ACCESS_REPORT – These are the tag reports from both inventory and access operations. Reports are delivered through the delegate **delegateRoAccessReport**.
- READER_EVENT_NOTIFICATION – These asynchronous events describe the Reader's status or health. One example includes the *AntennaEvent*, which reports when antennas are connected or disconnected. These reports are delivered via the delegate **delegateReaderEventNotification**.
- KEEPALIVE – These messages are sent by the Reader (configured by the client) and tell the Reader to periodically send messages to the client to validate connectivity. The Reader itself does not take action when these events are not delivered. These messages are delivered through the delegate **delegateKeepAlive**.

The following example shows installing the delegates at initialization.

LTKNET

```
#region EventHandlers
{
   Console.WriteLine("Adding Event Handlers\\n");
   reader.OnReaderEventNotification +=
      new delegateReaderEventNotification(reader_OnReaderEventNotification);
   reader.OnRoAccessReportReceived +=
```

```
      new delegateRoAccessReport(reader_OnRoAccessReportReceived);
}
#endregion
```

Figure 7.17 LTKNET –Adding Event Handlers for Asynchronous Reader Messages

The user-defined methods call for each message. In our simple example, we extract the EPC and print the hex value to the console output. Reports could be empty, so the application checks for empty reports, as shown in Figure 7.18. Note that, in general, more than one tag report can come in each message, even with the default set to immediate reporting.

LLRP has two different tag representations, a short format specific to 96-bit EPCs, and a general format suitable for all lengths of EPCs. We recommend that your application checks for both types of formats, and treats them appropriately as shown below.

LTKNET

```
// Simple Handler for receiving the tag reports from the reader
static void reader_OnRoAccessReportReceived(MSG_RO_ACCESS_REPORT msg)
{
   // Report could be empty
   if (msg.TagReportData == null) return;

   // Loop through and print out each tag
   for (int i = 0; i < msg.TagReportData.Length; i++)
   {
      reportCount++;
      // just write out the EPC as a hex string for now. It is guaranteed to be
      // in all LLRP reports regardless of default configuration
      string epc;
      if (msg.TagReportData[i].EPCParameter[0].GetType() ==
          typeof(PARAM_EPC_96))
      {
         epc = ((PARAM_EPC_96)
                (msg.TagReportData[i].EPCParameter[0])).EPC.ToHexString();
      }
      else
      {
         epc = ((PARAM_EPCData)
               (msg.TagReportData[i].EPCParameter[0])).EPC.ToHexString();
      }
      Console.WriteLine(epc);
```

```
    }
}
```

Figure 7.18 LTKNET –Receiving Asynchronous Reader Messages

7.7.3 LTKJava

In LTKJava, your application class inherits from **LLRPEndpoint**. **LLRPEnpoint** calls ***messageReceived*** whenever a message is received over the LLRP connection. Your class can override this implementation to handle received messages.

This function is responsible for dispatching the various types of messages (notifications, reports etc) that arrive on the connection. Thus, at the start of this function, check the message type. For RO_ACCESS_REPORT messages, we retrieve the tag list. The tag list is a standard Java list that can be iterated with standard methods shown below.

The tag report information can be extracted by examining the fields within the report.

Note: Most tag report data is optional, so it could be Null. In the example below, we simply fetch the data and format a string containing the tag information that is available.

LLRP has two different tag representations, a short format specific to 96-bit EPCs, and a general format suitable for all lengths of EPCs. We recommend that your application checks for both types of formats and treats them appropriately as shown below.

LTKJAVA

```
// messageReceived method is called whenever a message is received
// asynchronously on the LLRP connection.
public void messageReceived(LLRPMessage message)
{
    // convert all messages received to LTK-XML representation
    // and print them to the console
    logger.debug("Received " + message.getName() +
                 " message asynchronously");
    if (message.getTypeNum() == RO_ACCESS_REPORT.TYPENUM)
    {
        RO_ACCESS_REPORT report = (RO_ACCESS_REPORT) message;
        List<TagReportData> tdlist = report.getTagReportDataList();
        for (TagReportData tr : tdlist)
        {
            logOneTagReport(tr);
        }
```

```
    }
    else if (message.getTypeNum() == READER_EVENT_NOTIFICATION.TYPENUM)
    {
        // TODO
    }
}
```

Figure 7.19 LTKJava –Receiving Asynchronous Reader Messages

LTKJAVA

```
private void logOneTagReport(TagReportData tr)
{
    // As an example here, we'll just get the stuff out of here and
    // for a super long string
    LLRPParameter epcp = (LLRPParameter) tr.getEPCParameter();

    // epc is not optional, so we should fail if we can't find it
    String epcString = "unknownEPC";
    if(epcp != null) {
        if( epcp.getName().equals("EPC_96"))
        {
            EPC_96 epc96 = (EPC_96) epcp;
            epcString = epc96.getEPC().toString();
        }
        else if ( epcp.getName().equals("EPCData"))
        {
            EPCData epcData = (EPCData) epcp;
            epcString = epcData.getEPC().toString();
        }
    }
    else
    {
        logger.error("Could not find EPC in Tag Report");
        System.exit(1);
    }

    // all of these values are optional, so check their non-nullness first
    String antenna = "unknownAntenna";
    if(tr.getAntennaID() != null)
    {
```

```
      antenna = tr.getAntennaID().getAntennaID().toString();
   }
   String index = "unknownChannelIndex";
   if(tr.getChannelIndex() != null)
   {
      index = tr.getChannelIndex().getChannelIndex().toString();
   }
   String ftime = "unknownFirstTimestamp";
   if( tr.getFirstSeenTimestampUTC() != null)
   {
      ftime = tr.getFirstSeenTimestampUTC().getMicroseconds().toString();
   }
   String paramId = "unknownParamSpecID";
   if(tr.getInventoryParameterSpecID() != null)
   {
      paramId = tr.getInventoryParameterSpecID().
          getInventoryParameterSpecID().toString();
   }
   String ltime = "unknownLastTimestamp";
   if(tr.getLastSeenTimestampUTC() != null)
   {
      ltime = tr.getLastSeenTimestampUTC().getMicroseconds().toString();
   }
   String rssi = "unknownRssi";
   if(tr.getPeakRSSI() != null)
   {
      rssi = tr.getPeakRSSI().getPeakRSSI().toString();
   }
   String roid = "unknownRospecID";
   if(tr.getROSpecID() != null)
   {
      roid = tr.getROSpecID().getROSpecID().toString();
   }
   String tagseen = "unknownTagSeenCount";
   if(tr.getTagSeenCount() != null)
   {
      tagseen = tr.getTagSeenCount().getTagCount().toString();
   }
   String output = " EPC: " + epcString +
      ",Antenna: " + antenna +
```

```
      ",ChanIndex: " + index +
      ",SeenCnt: " + tagseen +
      ",FirstSeen: " + ftime +
      ",LastSeen: " + ltime +
      ",RSSI: " + rssi +
      ",ROSpecID: " + roid +
      ",ParamID: " + paramId;
   System.out.println(output);
}
```

Figure 7.20 LTKJava –Extracting Data From Tag Reports

8 Impinj Octane LTK Application

This section describes how to access the Octane Impinj LLRP extensions through the Impinj LTK. For a complete description of the Octane LLRP extensions, see the reference in Section 1.6.2. This example is included with the Impinj LTK as **docsample2.**

This example uses the SET_READER_CONFIG message to set the Reader's settings, and includes the Impinj settings for this use case.

An application wants to configure a two-antenna portal to track items flowing through the portal. To achieve peak performance, the system engineer will use the Impinj Gen2 AutoSet mode. To compensate for long cable runs over the large portal, the professional installer wants the Reader to operate at maximum power. The portal is lightly used, and the system engineer wants to ensure that the Reader is not using extra AC power or interfering with other RFID devices when no tags are present. The system is required to work on both FCC and ETSI Readers.

8.1 Analysis

The default value for the Speedway of 30 dBm transmits power that typically is dependent on the Regulatory region. Some versions of Speedway are capable of transmitting over 30 dBm to compensate for high losses in the cabling and antennas. Thus, the software in this example must set the max power based on the unit type. There are a few ways to accomplish this. You could search the data sheet for Readers in FCC and ETSI, and hard-code the LLRP power index by using a lookup table that is based on region and model number. Alternately, you could query the Reader for its transmit power capabilities. In this example we choose the latter. For the first approach, see Section 1.6.2 for a description of the document that contains this information.

The LLRP *RFTransmitte***r** parameter sets the transmit power. However, it also contains the *hopTableID* and *ChannelIndex*, which cannot be set independently. Instead, the application must query these three items for use in building the *RFTransmitte***r** parameter. The application must then send this parameter via the LLRP SET_READER_CONFIG message.

To conserve AC power and reduce interference, the application should configure the *ImpinjLowDutyCycle* as part of the SET_READER_CONFIG message. To use Impinj Autoset, the application must set the ***ModeID*** in the *C1G2RfControl* parameter.

For a detailed description of these Octane LLRP extensions, see Section 1.6.2.

Note: Because the LTKCPP example uses LTK-XML, this example is not available on Windows.

8.2 Application Flow

To implement the application flow use case, use the following steps:

1. Initialize the Library.
2. Connect to the Reader.
3. Enable Impinj Extensions.
4. Use the factory default LLRP configuration to ensure that the Reader is in a known state. (This simple example relies on the default Reader configuration.)
5. GET_READER_CAPABILITIES to learn the maximum power that is supported by this Reader.
6. GET_READER_CONFIG to get the *hopTableID* and *channelIndex* for use later when you set the *RFTransmitter* parameter.
7. SET_READER_CONFIG with the appropriate *InventorySearchMode*, *TagDirection*, **L***owDutyCycleMode*, *RFTransmitter*, *and AutoSet* values.
8. ADD_ROSPEC to tell the Reader to perform an inventory.
9. ENABLE_ROSPEC
10. START_ROSPEC starts the inventory operation.
11. Process RFID Data (EPC, RSSI, Timestamp) and direction data.

We have already seen steps 1-4, 9 and 10 in this document, so the example starts with step 5. For the complete sample code, see the **docSample2** directories in the Impinj LTK distribution.

For LTKCPP, we will provide the object based parameter and message construction. For LTKNET we will show the LTK-XML file-based construction when appropriate.

8.3 Impinj Capabilities/Getting Reader Capabilities

The LLRP GET_READER_CAPABILITIES_RESPONSE contains a lot of information on the features and functions supported by the Reader. In this example, we focus on learning the max power supported by the Reader and the Reader model number.

The transmit power levels are reported in a table within the *UHFBandCapabilities* parameter, which is in the GET_READER_CAPABILITIES_RESPONSE message. The table reports the Reader's available power levels. For each level, it reports an integer index to use when setting the power. In this example, we just extract the last entry in the table and retrieve its index max power.

The Reader *modelName* (a 32-bit number) is reported within the *GeneralDeviceCapabilities* parameter. Because this parameter might be used differently by different vendors, the application needs to check the *DeviceManufacturerName* and *modelName* combination to validate this parameter. For details on the Impinj manufacturer ID and model numbers, see Section 1.6.2.

In LTKNET, lists store as arrays. Look for the last element in the transmit power table and store its index and value.

LTKNET

```
#region getReaderCapabilities
{
   Console.WriteLine("Getting Reader Capabilities\\n");
   MSG_GET_READER_CAPABILITIES cap = new MSG_GET_READER_CAPABILITIES();
   cap.MSG_ID = 2; // Doesn't need to be set. The library will default
   cap.RequestedData = ENUM_GetReaderCapabilitiesRequestedData.All;

   //Send the custom message and wait for 8 seconds
   MSG_ERROR_MESSAGE msg_err;
   MSG_GET_READER_CAPABILITIES_RESPONSE msg_rsp =
      reader.GET_READER_CAPABILITIES(cap, out msg_err, 8000);
   if (msg_rsp != null)
   {
      if (msg_rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
      {
         Console.WriteLine(msg_rsp.LLRPStatus.StatusCode.ToString());
         reader.Close();
         return;
      }
   }
   else if (msg_err != null)
   {
      Console.WriteLine(msg_err.ToString());
      reader.Close();
      return;
   }
   else
   {
      Console.WriteLine("GET reader Capabilities Command Timed out\\n");
      reader.Close();
      return;
   }

   // Get the reader model number since some features are not
   // available on Speedway.
   PARAM_GeneralDeviceCapabilities dev_cap = msg_rsp.GeneralDeviceCapabilities;

   // Check to make sure the model number mathces and that this device
   // is an impinj reader (deviceManufacturerName == 25882)
   if ((dev_cap != null) &&
```

```
      (dev_cap.DeviceManufacturerName == 25882))
   {
      modelNumber = dev_cap.ModelName;
   }
   else
   {
      Console.WriteLine("Could not determine reader model number\\n");
      reader.Close();
      return;
   }

   // get the uhf band capabilities. Inside this is the power table.
   // take the last element of the power table to get the highest power.
   PARAM_UHFBandCapabilities uhf =
      msg_rsp.RegulatoryCapabilities.UHFBandCapabilities;
   PARAM_TransmitPowerLevelTableEntry entry =
      uhf.TransmitPowerLevelTableEntry[
        uhf.TransmitPowerLevelTableEntry.Length - 1];
   txPwrIndx = entry.Index;
   double power = entry.TransmitPowerValue / 100;
   Console.WriteLine(" Max Power " + power.ToString() + " dbm.");
}
#endregion
```

Figure 8.1 LTKNET – Retrieving Device Capabilities

For LTKCPP, the transmit power tables are stored as STL lists. The application needs to get the end of the list, back up one element to get the last valid element of the list, and then extract the index and power value from the table entry iterator.

The modelName and deviceManufacturerName are retrieved in a manner similar to the LTKNET example above.

LTKCPP

```
{
   CGET_READER_CAPABILITIES *pCmd;
   CMessage * pRspMsg;
   CGET_READER_CAPABILITIES_RESPONSE *pRsp;
   CRegulatoryCapabilities *pReg;
   CUHFBandCapabilities *pUhf;
   CTransmitPowerLevelTableEntry *pPwrLvl;
```

```
CGeneralDeviceCapabilities *pDeviceCap;
std::list<CTransmitPowerLevelTableEntry *>::iterator PwrLvl;

/* Compose the command message */
pCmd = new CGET_READER_CAPABILITIES();
pCmd->setMessageID(m_messageID++);
pCmd->setRequestedData(GetReaderCapabilitiesRequestedData_All);

/* Send the message, expect a certain type of response */
pRspMsg = transact(pCmd);

/* Done with the command message */
delete pCmd;

/* transact() returns NULL if something went wrong. */
if(NULL == pRspMsg)
{
   /* transact already tattled */
   return -1;
}
/* Cast to a CGET_READER_CAPABILITIES_RESPONSE message. */
pRsp = (CGET_READER_CAPABILITIES_RESPONSE *) pRspMsg;

/* Check the LLRPStatus parameter. */
if(0 != checkLLRPStatus(pRsp->getLLRPStatus(),
   "getReaderCapabilities"))
{
   /* checkLLRPStatus already tattled */
   delete pRspMsg;
   return -1;
}

/* Get out the Regulatory Capabilities element */
if(NULL == (pReg = pRsp->getRegulatoryCapabilities()))
{
   delete pRspMsg;
   return -1;
}

/* Get out the UHF Band Capabilities element */
```

```
if(NULL == (pUhf = pReg->getUHFBandCapabilities()))
{
   delete pRspMsg;
   return -1;
}

/* Get the last power level in the table */
PwrLvl = pUhf->endTransmitPowerLevelTableEntry();
PwrLvl--;

/* Store the index for use int the ROSpec */
pPwrLvl = *PwrLvl;
m_PowerLevelIndex = pPwrLvl->getIndex();
if(1 < m_Verbose)
{
   printf("INFO: Reader Max Power index %u, power %f\\n",
   pPwrLvl->getIndex(), pPwrLvl->getTransmitPowerValue()/100);
}

/* if this parameter is missing, or if this is not an Impinj
 reader, we can't determine its capabilities so we exit
 Impinj Private Enterprise NUmber is 25882 */
if( (NULL == (pDeviceCap = pRsp->getGeneralDeviceCapabilities())) \\
   (25882 != pDeviceCap->getDeviceManufacturerName()))
{
   delete pRspMsg;
   return -1;
}
m_modelNumber = pDeviceCap->getModelName();
if(1 < m_Verbose)
{
   printf("INFO: Reader Model Name %u\\n", m_modelNumber);
}

/* Done with the response message. */
delete pRspMsg;

/* Tattle progress, maybe */
if(m_Verbose)
{
```

```
        printf("INFO: Found LLRP Capabilities \\n");
    }

  /* Victory. */
    return 0;
}
```

Figure 8.2 LTKCPP – Retrieving Device Capabilities

In LTKJava, the power level entries are stored in a standard Java List object. The code in Figure 8.3 shows an example of validating the manufacturer and storing the device max power.

LTKJAVA

```
private void getReaderCapabilities()
{
   LLRPMessage response;
   GET_READER_CAPABILITIES_RESPONSE gresp;
   GET_READER_CAPABILITIES get = new GET_READER_CAPABILITIES();
   GetReaderCapabilitiesRequestedData data =
      new GetReaderCapabilitiesRequestedData(
   GetReaderCapabilitiesRequestedData.All);
   get.setRequestedData(data);
   get.setMessageID(getUniqueMessageID());
   logger.info("Sending GET_READER_CAPABILITIES message ...");
   try
   {
      response = connection.transact(get, 10000);
      // check whether GET_CAPABILITIES call was successful
      gresp = (GET_READER_CAPABILITIES_RESPONSE)response;
      StatusCode status = gresp.getLLRPStatus().getStatusCode();
      if (status.equals(new StatusCode("M_Success"))) {
         logger.info("GET_READER_CAPABILITIES was successful");
         // get the info we need
         GeneralDeviceCapabilities dev_cap = gresp.getGeneralDeviceCapabilities();
         if ((dev_cap == null) \\
             (!dev_cap.getDeviceManufacturerName().equals(new
                   UnsignedInteger(25882)))) {
             logger.error("DocSample2 must use Impinj model Reader, not " +
             dev_cap.getDeviceManufacturerName().toString());
             System.exit(1);
```

```
            }
            modelName = dev_cap.getModelName();
            logger.info("Found Impinj reader model " + modelName.toString());
            // get the max power level
            if( gresp.getRegulatoryCapabilities() != null) {
               UHFBandCapabilities band_cap =
                  gresp.getRegulatoryCapabilities().getUHFBandCapabilities();
               List<TransmitPowerLevelTableEntry> pwr_list =
                  band_cap.getTransmitPowerLevelTableEntryList();
               TransmitPowerLevelTableEntry entry =
                  pwr_list.get(pwr_list.size() - 1);
               maxPowerIndex = entry.getIndex();
               maxPower = entry.getTransmitPowerValue();
               // LLRP sends power in dBm * 100
               double d = ((double) maxPower.intValue())/100;
               logger.info("Max power " + d +
                  " dBm at index " + maxPowerIndex.toString());
            }
         }
         else {
            logger.info(response.toXMLString());
            logger.info("GET_READER_CAPABILITIES failures");
            System.exit(1);
         }
      } catch (InvalidLLRPMessageException ex) {
         logger.error("Could not display response string");
      } catch (TimeoutException ex) {
         logger.error("Timeout waiting for GET_READER_CAPABILITIES response");
         System.exit(1);
      }
}
```

Figure 8.3 LTKJAVA – Retrieving Device Capabilities

8.4 Get Reader Configuration

As mentioned above, setting the *RFTransmitter* parameter that contains the transmit power level also requires setting *hopTableID* and *channelIndex*. These can be retrieved via the GET_READER_CONFIG message. This message returns one configuration set for each antenna supported on the Speedway. Because the default channel for Speedway is the same value for all

antennas, the application can retrieve the *hopTableID* and *channelIndex* for the first antenna only. Both of these parameters reside within the RFTransmmitter parameter, which is inside the *AntennaConfiguration* parameter.

Note: This is a known issue with LLRP. Impinj is working with EPCglobal to correct this problem in future LLRP versions.

LTKNET

```
{
   Console.WriteLine("Getting Reader Configuration\\n");
   MSG_GET_READER_CONFIG cap = new MSG_GET_READER_CONFIG();
   cap.MSG_ID = 2; // Note: This doesn't need to be set. The library will default
   cap.RequestedData = ENUM_GetReaderConfigRequestedData.All;
   //Send the custom message and wait for 8 seconds
   MSG_ERROR_MESSAGE msg_err;
   MSG_GET_READER_CONFIG_RESPONSE msg_rsp =
   reader.GET_READER_CONFIG(cap, out msg_err, 8000);
   if (msg_rsp != null)
   {
      if (msg_rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
      {
         Console.WriteLine(msg_rsp.LLRPStatus.StatusCode.ToString());
         reader.Close();
         return;
      }
   }
   else if (msg_err != null)
   {
      Console.WriteLine(msg_err.ToString());
      reader.Close();
      return;
   }
   else
   {
      Console.WriteLine("GET reader Config Command Timed out\\n");
      reader.Close();
      return;
   }
   // Get the hopTableId and Channel Index
   if ((null != msg_rsp.AntennaConfiguration)
      && (0 < msg_rsp.AntennaConfiguration.Length)
```

```
      && (null != msg_rsp.AntennaConfiguration[0].RFTransmitter))
   {
      PARAM_RFTransmitter rftx =
         msg_rsp.AntennaConfiguration[0].RFTransmitter;
      // we have to get these two values as well otherwise
      // we won't know what to fill in the RFTransmitter
      // parameter when we set transmit power
      ChannelIndex = rftx.ChannelIndex;
      hopTableID = rftx.HopTableID;
      Console.WriteLine(" Saving ChanIndex " +
         ChannelIndex.ToString() +
         " hopTableId " +
         hopTableID.ToString());
   }
   else
   {
      Console.WriteLine("Could not get rf transmitter parameters\\n");
      reader.Close();
      return;
   }
}
```

LTKCPP

```
{
   CGET_READER_CONFIG *pCmd;
   CMessage * pRspMsg;
   CGET_READER_CONFIG_RESPONSE *pRsp;
   std::list<CAntennaConfiguration*>::iterator pAntCfg;

   /* Compose the command message */
   pCmd = new CGET_READER_CONFIG();
   pCmd->setMessageID(m_messageID++);
   pCmd->setRequestedData(GetReaderConfigRequestedData_All);

   /* Send the message, expect a certain type of response */
   pRspMsg = transact(pCmd);

   /* Done with the command message */
   delete pCmd;
```

```
/* transact() returns NULL if something went wrong. */
if(NULL == pRspMsg)
{
   /* transact already tattled */
   return -1;
}

/* Cast to a CGET_READER_CONFIG_RESPONSE message. */
pRsp = (CGET_READER_CONFIG_RESPONSE *) pRspMsg;

/* Check the LLRPStatus parameter. */
if(0 != checkLLRPStatus(pRsp->getLLRPStatus(),
   "getReaderConfig"))
{
   /* checkLLRPStatus already tattled */
   delete pRspMsg;
   return -1;
}

/* just get the hop table and channel index out of
 the first antenna configuration since they must all
 be the same */
pAntCfg = pRsp->beginAntennaConfiguration();
if(pAntCfg != pRsp->endAntennaConfiguration())
{
   CRFTransmitter *prfTx;
   prfTx = (*pAntCfg)->getRFTransmitter();
   m_hopTableID = prfTx->getHopTableID();
   m_channelIndex = prfTx->getChannelIndex();
}
else
{
   delete pRspMsg;
  return -1;
}
if(1 < m_Verbose)
{
   printf("INFO: Reader hopTableID %u, ChannelIndex %u\\n",
   m_hopTableID, m_channelIndex);
}
```

```
    /* Done with the response message. */
    delete pRspMsg;

    /* Tattle progress, maybe */
    if(m_Verbose)
    {
        printf("INFO: Found LLRP Configuration \\n");
    }

    /* Victory. */
    return 0;
}
```

LTKJAVA

```
private void getReaderConfiguration() {
    LLRPMessage response;
    GET_READER_CONFIG_RESPONSE gresp;
    GET_READER_CONFIG get = new GET_READER_CONFIG();
    GetReaderConfigRequestedData data =
        new GetReaderConfigRequestedData(
            GetReaderConfigRequestedData.All);
    get.setRequestedData(data);
    get.setMessageID(getUniqueMessageID());
    get.setAntennaID(new UnsignedShort(0));
    get.setGPIPortNum(new UnsignedShort(0));
    get.setGPOPortNum(new UnsignedShort(0));
    logger.info("Sending GET_READER_CONFIG message ...");

    try {
        response = connection.transact(get, 10000);
        // check whether GET_CAPABILITIES addition was successful
        gresp = (GET_READER_CONFIG_RESPONSE)response;
        StatusCode status = gresp.getLLRPStatus().getStatusCode();

        if (status.equals(new StatusCode("M_Success"))) {
            logger.info("GET_READER_CONFIG was successful");
            List<AntennaConfiguration> alist = gresp.getAntennaConfigurationList();
            if(!alist.isEmpty()) {
                AntennaConfiguration a_cfg = alist.get(0);
```

```
            channelIndex = a_cfg.getRFTransmitter().getChannelIndex();
            hopTableID = a_cfg.getRFTransmitter().getHopTableID();
            logger.info("ChannelIndex " + channelIndex.toString() +
               " hopTableID " + hopTableID.toString());
         } else {
            logger.error("Could not find antenna configuration");
            System.exit(1);
         }
      }
      else {
         logger.info(response.toXMLString());
         logger.info("GET_READER_CONFIG failures");
         System.exit(1);
      }

   } catch (InvalidLLRPMessageException ex) {
      logger.error("Could not display response string");
   } catch (TimeoutException ex) {
      logger.error("Timeout waiting for GET_READER_CONFIG response");
      System.exit(1);
   }
}
```

8.5 Impinj Configuration

Many Impinj Octane extensions can be set via the SET_READER_CONFIG message. In this example, we use LTK-XML with LTKNET and LTKCPP to apply the following configuration. The configuration specifics in the example are addressed via a single SET_READER_CONFIG message with the following LTK-XML.

8.5.1 LTK-XML

LTK-XML

```
<?xml version="1.0" encoding="utf-8" ?>

 <SET_READER_CONFIG
    xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
    xmlns:llrp="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.6"
    xsi:schemaLocation="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0
    http://www.llrp.org/ltk/schema/core/encoding/xml/1.0/llrp.xsd
    http://developer.impinj.com/ltk/schema/encoding/xml/1.6
    http://developer.impinj.com/ltk/schema/encoding/xml/1.6/impinj.xsd"

     MessageID="0">
       <ResetToFactoryDefault>false</ResetToFactoryDefault>
           <AntennaConfiguration>
             <AntennaID>0</AntennaID>
             <!-- we will over-write the transmit power in our code -->
             <RFTransmitter>
                 <HopTableID>1</HopTableID>
                 <ChannelIndex>1</ChannelIndex>
                 <TransmitPower>1</TransmitPower>
             </RFTransmitter>
             <C1G2InventoryCommand>
                 <TagInventoryStateAware>false</TagInventoryStateAware>
                 <C1G2RFControl>
                   <!--Set mode to Gen2 DRM AutoSet Mode Tari is ignored -->
                   <ModeIndex>1000</ModeIndex>
                   <Tari>0</Tari>
                 </C1G2RFControl>
                 <C1G2SingulationControl>
                   <!--Will use session 2 -->
                   <Session>2</Session>
                   <TagPopulation>32</TagPopulation>
                   <TagTransitTime>0</TagTransitTime>
                 </C1G2SingulationControl>
                 <Impinj:ImpinjInventorySearchMode xmlns=
                     "http://developer.impinj.com/ltk/schema/encoding/xml/1.6">
                   <!--Will use Dual-target-->
                   <InventorySearchMode>Dual_Target</InventorySearchMode>
                 </Impinj:ImpinjInventorySearchMode>

                 <!--Enable Low Duty Cycle when no tags are seen for 10 seconds.
                         Check antennas every 200 msec -->
             <Impinj:ImpinjLowDutyCycle xmlns=
             "http://developer.impinj.com/ltk/schema/encoding/xml/1.6">
```

```
                <LowDutyCycleMode>Enabled</LowDutyCycleMode>
                <EmptyFieldTimeout>10000</EmptyFieldTimeout>
                <FieldPingInterval>200</FieldPingInterval>
            </Impinj:ImpinjLowDutyCycle>
            </C1G2InventoryCommand>
          </AntennaConfiguration>
          <ROReportSpec>
          <ROReportTrigger>Upon_N_Tags_Or_End_Of_ROSpec
          </ROReportTrigger>
          <N>1</N>
          <TagReportContentSelector>
            <EnableROSpecID>false</EnableROSpecID>
            <EnableSpecIndex>false</EnableSpecIndex>
            <EnableInventoryParameterSpecID>false
            </EnableInventoryParameterSpecID>
            <EnableAntennaID>false</EnableAntennaID>
            <EnableChannelIndex>false</EnableChannelIndex>
            <EnablePeakRSSI>false</EnablePeakRSSI>
            <EnableFirstSeenTimestamp>false</EnableFirstSeenTimestamp>
            <EnableLastSeenTimestamp>false</EnableLastSeenTimestamp>
            <EnableTagSeenCount>false</EnableTagSeenCount>
            <EnableAccessSpecID>false</EnableAccessSpecID>
            <C1G2EPCMemorySelector>
                <EnableCRC>false</EnableCRC>
                <EnablePCBits>false</EnablePCBits>
            </C1G2EPCMemorySelector>
          </TagReportContentSelector>
          <!-- Don't need any extra tag information beyond EPC -->
      </ROReportSpec>
    </SET_READER_CONFIG>
```

Figure 8.4 LTK-XML – Impinj Octane LLRP Extensions

8.5.2 LTKNET

The C# code that provisions this LTKNET configuration on the Reader Gateway is shown in Figure 8.5. Note: this is nearly identical to the *RoSpec* example in Section 7.4.2.

```
#region SetReaderConfigWithXML
{
   Console.WriteLine("Adding SET_READER_CONFIG from XML file \\n");
   Org.LLRP.LTK.LLRPV1.DataType.Message obj;
   ENUM_LLRP_MSG_TYPE msg_type;

   // read the XML from a file and validate its an ADD_ROSPEC
   try
   {
      FileStream fs = new FileStream(@"..\\..\\setReaderConfig.xml",
         FileMode.Open);
      StreamReader sr = new StreamReader(fs);
      string s = sr.ReadToEnd();
      fs.Close();
      LLRPXmlParser.ParseXMLToLLRPMessage(s, out obj, out msg_type);
      if (obj == null \\ msg_type !=

         ENUM_LLRP_MSG_TYPE.SET_READER_CONFIG)
      {
         Console.WriteLine("Could not extract message from XML");
         reader.Close();
         return;
      }
   }
   catch
   {
      Console.WriteLine("Unable to convert to valid XML");
      reader.Close();
      return;
   }
   // Communicate that message to the reader
   MSG_SET_READER_CONFIG msg = (MSG_SET_READER_CONFIG)obj;
   // set the max power available but don't forget to
   // apply the hoptable and channelIndex from the
   // current configuration
   PARAM_AntennaConfiguration ant = msg.AntennaConfiguration[0];
   ant.RFTransmitter.TransmitPower = (ushort)txPwrIndx;
   ant.RFTransmitter.ChannelIndex = (ushort)ChannelIndex;
   ant.RFTransmitter.HopTableID = (ushort)hopTableID;
   MSG_ERROR_MESSAGE msg_err;
```

```
    MSG_SET_READER_CONFIG_RESPONSE rsp =

        reader.SET_READER_CONFIG(msg, out msg_err, 12000);
    if (rsp != null)
    {
        if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
        {
            Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
            reader.Close();
            return;
        }
    }
    else if (msg_err != null)
    {
        Console.WriteLine(msg_err.ToString());
        reader.Close();
        return;
    }
    else
    {
        Console.WriteLine("SET_READER_CONFIG Command Timed out\\n");
        reader.Close();
        return;
    }
}
#endregion
```

Figure 8.5 LTKNET – Setting Configuration with LTK-XML

8.5.3 LTKCPP

To set an advanced configuration within a **CSET_READER_CONFIG**, the example in Figure 8.6 uses LTK-XML to create a **CSET_READER_CONFIG** object.

```
{
    CMessage * pCmdMsg;
    CSET_READER_CONFIG *pCmd;
    CMessage * pRspMsg;
    CSET_READER_CONFIG_RESPONSE *pRsp;
```

```
    CXMLTextDecoder * pDecoder;
    std::list<CAntennaConfiguration *>::iterator Cur;

    /* Build a decoder to extract the message from XML */
    pDecoder = new CXMLTextDecoder(m_pTypeRegistry, "setReaderConfig.xml");
    if(NULL == pDecoder)
    {
        return -1;
    }

    pCmdMsg = pDecoder->decodeMessage();
    delete pDecoder;
    if(NULL == pCmdMsg)
    {
        return -2;
    }
    if(&CSET_READER_CONFIG::s_typeDescriptor != pCmdMsg->m_pType)
    {
        return -3;
    }

    /* get the message as a SET_READER_CONFIG */
    pCmd = (CSET_READER_CONFIG *) pCmdMsg;

    /* It's always a good idea to give it a unique message ID */
    pCmd->setMessageID(m_messageID++);
```

Figure 8.6 LTKCPP – Importing SET_READER_CONFIG from LTK-XML File

Modifying the transmit power requires the application to access the **CAntennaConfiguration** objects within this message. Then it creates or modifies the **CRFTransmitter** object and sets the power level index to the value determined in section 8.3.

```
/* at this point, we would be ready to send the message, but we need
* to make a change to the transmit power for each enabled antenna.
* Loop through */
for(
    Cur = pCmd->beginAntennaConfiguration();
    Cur != pCmd->endAntennaConfiguration();
    Cur++)
{
```

```
    CRFTransmitter *pRfTx = (*Cur)->getRFTransmitter();

    /* we already have this element in our sample XML file, but
    * we check here to create one if it doesn't exist to show
    * a more general usage */
    if(NULL == pRfTx)
    {
       pRfTx = new CRFTransmitter();
       (*Cur)->setRFTransmitter(pRfTx);
    }

    /*
     Set the max power that we retreived from the capabilities
     and the hopTableID and Channel index we got from the config
    */
    pRfTx->setChannelIndex(m_channelIndex);
    pRfTx->setHopTableID(m_hopTableID);
    pRfTx->setTransmitPower(m_PowerLevelIndex);
 }

/* Send the message, expect a certain type of response */
pRspMsg = transact(pCmd);

/* Done with the command message */
delete pCmd;

/* transact() returns NULL if something went wrong. */
if(NULL == pRspMsg)
{
   /* transact already tattled */
   return -1;
}

/* Cast to a CSET_READER_CONFIG_RESPONSE message. */
pRsp = (CSET_READER_CONFIG_RESPONSE *) pRspMsg;

/* Check the LLRPStatus parameter. */
if(0 != checkLLRPStatus(pRsp->getLLRPStatus(),
   "setImpinjReaderConfig"))
{
```

```
    /* checkLLRPStatus already tattled */
    delete pRspMsg;
    return -1;
}

/* Done with the response message. */
delete pRspMsg;

/* Tattle progress, maybe */
if (m_Verbose)
{
    printf("INFO: Set Impinj Reader Configuration \\n");
}

/* Victory. */
return 0;
```

Figure 8.7 LTKCPP – Setting Antenna Configuration and Transmit Power

8.5.4 LTKJAVA

This section provides the Java code to use to provision this configuration. Note specifically how the XML reads and exceptions are handled. In addition, notice how to access the parameter fields, and how to add a custom parameter.

LTKJAVA

```
private void setReaderConfiguration()
{
    LLRPMessage response;
    logger.info("Loading SET_READER_CONFIG message from file
        SET_READER_CONFIG.xml ...");

    try
    {
        LLRPMessage setConfigMsg = Util.loadXMLLLRPMessage(new File("./source/org/
            impinj/llrp/ltk/examples/docsample2/SET_READER_CONFIG.xml"));

        // TODO make sure this is an SET_READER_CONFIG message
        // touch up the transmit power for max
```

```
    SET_READER_CONFIG setConfig = (SET_READER_CONFIG) setConfigMsg;
    AntennaConfiguration a_cfg = setConfig.getAntennaConfigurationList().
        get(0);
    RFTransmitter rftx = a_cfg.getRFTransmitter();
    rftx.setChannelIndex(channelIndex);
    rftx.setHopTableID(hopTableID);
    rftx.setTransmitPower(maxPowerIndex);
    response = connection.transact(setConfig, 10000);

    // check whetherSET_READER_CONFIG addition was successful
    StatusCode status = ((SET_READER_CONFIG_RESPONSE)response).
        getLLRPStatus().getStatusCode();

    if (status.equals(new StatusCode("M_Success")))
    {
       logger.info("SET_READER_CONFIG was successful");
    }
    else
    {
       logger.info(response.toXMLString());
       logger.info("SET_READER_CONFIG failures");
       System.exit(1);
    }

 }
 catch (TimeoutException ex)
 {
    logger.error("Timeout waiting for SET_READER_CONFIG response");
    System.exit(1);
 }
 catch (FileNotFoundException ex)
 {
    logger.error("Could not find file");
    System.exit(1);
 }
 catch (IOException ex)
 {
    logger.error("IO Exception on file");
    System.exit(1);
 } catch (JDOMException ex)
```

```
    {
        logger.error("Unable to convert LTK-XML to DOM");
        System.exit(1);
    }
    catch (InvalidLLRPMessageException ex)
    {
        logger.error("Unable to convert LTK-XML to Internal Object");
        System.exit(1);
    }
}
```

Figure 8.8 LTKJAVA – Setting Antenna Configuration and Transmit Power

9 LLRP Access, Filtering, and Accumulation

In this section we review an example that uses advanced features of LLRP and the Impinj LTK for controlling the flow and contents of tag reports. This example is included with the Impinj LTK as '**docsample3**'.

For this example, consider the following use case:

> The customer wants to read tags through a portal separating a restricted storage facility from their main manufacturing facility. The facility is located off-site and connects to the enterprise by a low-bandwidth link. The RFID system serves as an audit tool and will not be used for restricted access control. Personnel and asset observations must be time-stamped to make an accurate correlation between them.
>
> Two types of tags will pass through the portal: assets and personnel tags. For asset tags, all of the information is contained in the EPC. For personnel tags, however, the Gen2 user memory contains a code that describes their access permissions for the facility and company affiliation, and will be used off-line for equipment auditing. Personnel tags use a 96-bit GID EPC format, while assets use a 96-bit GIAI format. The facility is shared and contains several other RFID applications.
>
> Data is to be sent over a very low-bandwidth connection. Accumulation should be used to report tags only once every 30 seconds at most. The report size should be minimized.
>
> Because the Reader is located at a share facility, the Reader should minimize its RF interference.

9.1 Analysis

The application uses the Speedway LLRP accumulation feature and limits the quantity and contents of the tag reports from the Reader.

The application uses the LLRP C1G2Filter and eliminates reads of all tags except for GID-96 and GRAI-96.

Because the Reader should minimize its RF interference, use only low duty cycle mode as described in Section 8.5.

9.2 Application Flow

The steps required to implement the above use case are:

1. Initialize the Library.

2. Connect to the Reader.
3. Enable Impinj Extensions.
4. Use the factory default LLRP configuration to ensure that the Reader is in a known state. (We are relying on the default Reader configuration for this simple example.)
5. SET_READER_CONFIG with the appropriate settings for report generation as well as Impinj Low Duty Cycle mode, to reduce interference.
6. ADD_ROSPEC to instruct the Reader to perform an inventory. Include tag filters to reduce unwanted reads from other RFID applications
7. ADD_ACCESSSPEC to instruct the Reader to read user memory for personnel tags.
8. ENABLE_ROSPEC
9. ENABLE_ACCESSSPEC
10. START_ROSPEC starts the inventory operation.
11. Use GET_REPORT for RFID data and to process RFID Data (EPC, and Timestamp).

Steps 1-5, 7, 9, and 11 were covered in earlier examples. Examples for steps 6, 8 and 10 are included in this section.

9.3 Report Accumulation

The *ROReportSpec* contains the rules that define when a report is generated automatically by the Reader, and also what information the report contains. Rules for what data to report are set via the *tagReportContentSelector*. Since bandwidth is restricted, the application enables only the *firstSeemTimestamp* within the *tagReportContentSelecto***r**. This requires an accurate association of the assets observations with the personnel observations. EPCs are automatically provided and cannot be disabled.

The *ROReportTrigger* within *ROReportSpec* selects when the Reader will automatically generate reports. For this use case, the application sets the report trigger to NULL and polls for reports every few seconds.

> **Impinj Best Practice!:** *An alternate method is to have the Reader generate periodic reports automatically. The application should configure the ROSpec to start immediately and stop after 30 seconds, and then set the report trigger to trigger at the end of a ROSpec execution. This causes an asynchronous report to be sent every 30 seconds. Although this uses slightly less bandwidth than the polling case above, we use polling in our example because it is versatile for more use cases, and because it re-uses the ROSpec from previous examples.*

9.3.1 LTKNET

In LTKNET, the example uses a similar SET_READER_CONFIG message as in **docsample2** (Section 8.5.2), along with the same supporting code to provision the Reader settings, except that we use the default values for the Gen2 parameters. Enabling the timestamp allows the application to correlate GIDs and GRAIs.

LTK-XML

```
<?xml version="1.0" encoding="utf-8" ?>
<SET_READER_CONFIG
   xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:llrp="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.6"
   xsi:schemaLocation="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0
   http://www.llrp.org/ltk/schema/core/encoding/xml/1.0/llrp.xsd
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6/impinj.xsd"

MessageID="0">

<ResetToFactoryDefault>false</ResetToFactoryDefault>
<AntennaConfiguration>
    <AntennaID>0</AntennaID>
    <C1G2InventoryCommand>
      <TagInventoryStateAware>false</TagInventoryStateAware>
      <!--Enable Low Duty Cycle when no tags are seen for 10 seconds.
      Check antennas every 200 msec -->
      <Impinj:ImpinjLowDutyCycle xmlns="http://developer.impinj.com/ltk/schema/
      encoding/xml/1.6">
          <LowDutyCycleMode>Enabled</LowDutyCycleMode>
          <EmptyFieldTimeout>10000</EmptyFieldTimeout>
          <FieldPingInterval>200</FieldPingInterval>
      </Impinj:ImpinjLowDutyCycle>
    </C1G2InventoryCommand>
</AntennaConfiguration>
<ROReportSpec>
    <ROReportTrigger>None</ROReportTrigger>
    <N>0</N>
    <TagReportContentSelector>
      <EnableROSpecID>false</EnableROSpecID>
```

```
      <EnableSpecIndex>false</EnableSpecIndex>
      <EnableInventoryParameterSpecID>false</EnableInventoryParameterSpecID>
      <EnableAntennaID>false</EnableAntennaID>
      <EnableChannelIndex>false</EnableChannelIndex>
      <EnablePeakRSSI>false</EnablePeakRSSI>
      <EnableFirstSeenTimestamp>true</EnableFirstSeenTimestamp>
      <EnableLastSeenTimestamp>false</EnableLastSeenTimestamp>
      <EnableTagSeenCount>false</EnableTagSeenCount>
      <EnableAccessSpecID>false</EnableAccessSpecID>
      <C1G2EPCMemorySelector>
          <EnableCRC>false</EnableCRC>
          <EnablePCBits>false</EnablePCBits>
      </C1G2EPCMemorySelector>
    </TagReportContentSelector>
</ROReportSpec>
</SET_READER_CONFIG>
```

Figure 9.1 LTK-XML – SET_READER_CONFIG for docSample3

9.3.2 LTKCPP

In LTKCPP, the example in Figure 9.2 uses a code similar to **docsample2** (Section 8.5.3), except it uses the default values for the Gen2 parameters.

LTKCPP

```
/* Apply this configuration to all antennas */
pAnt->setAntennaID(0);

/* Create the container Inventory command to hold all the parameters */
CC1G2InventoryCommand *pC1G2Inv = new CC1G2InventoryCommand();

/* set the Impinj Low Duty Cycle mode as per the use case */
CImpinjLowDutyCycle *pImpLdc = new CImpinjLowDutyCycle();
pImpLdc->setEmptyFieldTimeout(10000);
pImpLdc->setFieldPingInterval(200);
pImpLdc->setLowDutyCycleMode(ImpinjLowDutyCycleMode_Enabled);
pC1G2Inv->addCustom(pImpLdc);

/*
```

```
 Don't forget to add the InventoryCommand to the antenna
 configuration, and then add the antenna configuration to the
 config message
*/
pAnt->addAirProtocolInventoryCommandSettings(pC1G2Inv);
pCmd->addAntennaConfiguration(pAnt);

/* report every tag (N=1) */
CROReportSpec *pROrs = new CROReportSpec();
pROrs->setROReportTrigger(ROReportTriggerType_None);
pROrs->setN(0);

/*
 Turn off report data that we don't need since our use
 case suggests we are bandwidth constrained
*/
CTagReportContentSelector *pROcontent = new CTagReportContentSelector();
pROcontent->setEnableAccessSpecID(false);
pROcontent->setEnableAntennaID(false);
pROcontent->setEnableChannelIndex(false);
pROcontent->setEnableFirstSeenTimestamp(true);
pROcontent->setEnableInventoryParameterSpecID(false);
pROcontent->setEnableLastSeenTimestamp(false);
pROcontent->setEnablePeakRSSI(false);
pROcontent->setEnableROSpecID(false);
pROcontent->setEnableSpecIndex(false);
pROcontent->setEnableTagSeenCount(false);
CC1G2EPCMemorySelector *pC1G2Mem = new CC1G2EPCMemorySelector();
pC1G2Mem->setEnableCRC(false);
pC1G2Mem->setEnablePCBits(false);
pROcontent->addAirProtocolEPCMemorySelector(pC1G2Mem);
pROrs->setTagReportContentSelector(pROcontent);
pCmd->setROReportSpec(pROrs);

/* Send the message, expect a certain type of response */
pRspMsg = transact(pCmd);
```

Figure 9.2 LTKCPP – SET_READER_CONFIG for docSample3

9.3.3 LTKJAVA

In LTKJAVA, the example uses a similar SET_READER_CONFIG message as **docsample2**, along with the same supporting code to provision these settings on the Reader, except that it uses the default values for the Gen2 parameters. Enabling the timestamp allows the application to correlate GIDs and GRAIs.

9.4 Tag Filtering

The use case requires the Reader to filter out all codes that are not GID-96 and GRAI-96. Because the system must preserve link bandwidth, we add *C1G2Filters* to the Reader, which filters out tags that don't match these patterns. To build this spec, review the details of the GID-96 and GRAI-96 formats.

EPCglobal defines the first byte of the EPC as the EPC header. This contains information about the type of EPC stored in the remaining bytes. The header for a GID-96 is **0b00110101**. The header for a GRAI-96 is **0b00110011**. Thus, in bits, look for an EPC sequence that starts with either of these values.

In the Gen2 tag protocol specification, the EPC memory starts at bit offset 0x20 in the EPC memory bank (bank 1) of the tag.

> **Impinj Best Practice!** *The PC bits contain the EPC length and are a good "double check" for reading the EPC header. However, the PC bits also contain several other indicators (UMI, XPC etc) that are not guaranteed to be the same across all tag variants. If your application filters on the PC bits, be sure to only filter on the first 5 bits (0x10 – 0x14).*

In LLRP, we add two *C1G2Filter* parameters to our *ROSpec* to select only these tags for inventory. The filter *Action* uses the Select_Unselect action for the first filter, which excludes (unselects) all tags that don't match the filter (GRAI) and includes (selects) all tags that do match the filter. The second filter uses the Select_DoNothing action, which builds on the first filter by adding those tags that match the GID filter (selects) and leaving the other tags unchanged (doNothing). If we re-used the Select_Unselect filter on the second filter, it would have unselected the GRAI tags from the first filter.

9.4.1 LTKNET

In LTKNET, the LTK-XML uses construction based on the *ROSpec*. Figure 9.3 shows an excerpt of the *ROSpec* with the *C1G2Filters* added to the *C1G2InventoryCommand*.

LTK-XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<AntennaConfiguration>
  <AntennaID>0</AntennaID>
  <C1G2InventoryCommand>
      <TagInventoryStateAware>false</TagInventoryStateAware>
      <C1G2Filter>
        <T>Do_Not_Truncate</T>
        <C1G2TagInventoryMask>
            <MB>1</MB>
            <Pointer>32</Pointer>
            <!-- pointer is in decimal -->
            <TagMask Count="8">33</TagMask>
        </C1G2TagInventoryMask>
        <C1G2TagInventoryStateUnawareFilterAction>
            <Action>Select_Unselect</Action>
        </C1G2TagInventoryStateUnawareFilterAction>
      </C1G2Filter>
      <C1G2Filter>
        <T>Do_Not_Truncate</T>
        <C1G2TagInventoryMask>
            <MB>1</MB>
            <Pointer>32</Pointer>
            <!-- pointer is in decimal -->
            <TagMask Count="8">35</TagMask>
        </C1G2TagInventoryMask>
        <C1G2TagInventoryStateUnawareFilterAction>
            <Action>Select_DoNothing</Action>
        </C1G2TagInventoryStateUnawareFilterAction>
      </C1G2Filter>
  </C1G2InventoryCommand>
</AntennaConfiguration>
```

Figure 9.3 LTKNET – InventoryFilters for docSample3

This *AntennaConfiguration* element is included in the *ROSpec* and is added by using code that is similar to Section 8.5.2.

9.4.2 LTKCPP

In LTKCPP, we build the *CInventoryParameterSpec* object and all of its sub-parameters.

Note: The pointers and lengths specified by *C1G2TagInventoryMask* are specified in bits.

LTKCPP

```
CInventoryParameterSpec * pInventoryParameterSpec =
    new CInventoryParameterSpec();
pInventoryParameterSpec->setInventoryParameterSpecID(1234);
pInventoryParameterSpec->setProtocolID(AirProtocols_EPCGlobalClass1Gen2);

/* make the bit pattern for the GID mask */
llrp_u1v_t gidMask = llrp_u1v_t(8);
gidMask.m_nBit = 8;
gidMask.m_pValue[0] = 0x33;

/* build the mask for the GID */
CC1G2TagInventoryMask *pMaskGID = new(CC1G2TagInventoryMask);
pMaskGID->setMB(1);
pMaskGID->setPointer(32);
pMaskGID->setTagMask(gidMask);

/* build the inventory action for the GID filter */
CC1G2TagInventoryStateUnawareFilterAction *pActionGID=
    new CC1G2TagInventoryStateUnawareFilterAction();
pActionGID->setAction(C1G2StateUnawareAction_Select_Unselect);

/* Build the filter for the GID */
CC1G2Filter *pFilterGID = new CC1G2Filter();
pFilterGID->setC1G2TagInventoryStateUnawareFilterAction(pActionGID);
pFilterGID->setC1G2TagInventoryMask(pMaskGID);
pFilterGID->setT(C1G2TruncateAction_Do_Not_Truncate);

/* make the bit pattern for the GRAI mask */
llrp_u1v_t graiMask = llrp_u1v_t(8);
graiMask.m_nBit = 8;
graiMask.m_pValue[0] = 0x35;

/* build the mask for the GRAI */
CC1G2TagInventoryMask *pMaskGRAI = new(CC1G2TagInventoryMask);
pMaskGRAI->setMB(1);
pMaskGRAI->setPointer(32);
pMaskGRAI->setTagMask(graiMask);
```

```
/* build the inventory action for the FRAI filter */
CC1G2TagInventoryStateUnawareFilterAction *pActionGRAI=
   new CC1G2TagInventoryStateUnawareFilterAction();
pActionGRAI->setAction(C1G2StateUnawareAction_Select_DoNothing);

/* Build the filter for the GRAI */
CC1G2Filter *pFilterGRAI = new CC1G2Filter();
pFilterGRAI->setC1G2TagInventoryStateUnawareFilterAction(pActionGRAI);
pFilterGRAI->setC1G2TagInventoryMask(pMaskGRAI);
pFilterGRAI->setT(C1G2TruncateAction_Do_Not_Truncate);

/* build the inventory command and add both filters */
CC1G2InventoryCommand *pInvCmd = new CC1G2InventoryCommand();
pInvCmd->setTagInventoryStateAware(false);
pInvCmd->addC1G2Filter(pFilterGID);
pInvCmd->addC1G2Filter(pFilterGRAI);

/* Build the antennaConfiguration to Contain this */
CAntennaConfiguration * pAntennaConfiguration =
   new CAntennaConfiguration();
pAntennaConfiguration->setAntennaID(0);
pAntennaConfiguration->addAirProtocolInventoryCommandSettings(pInvCmd);

/* don't forget to add this to the INventory Parameter Spec above */
pInventoryParameterSpec->addAntennaConfiguration(pAntennaConfiguration);
```

9.4.3 LTKJAVA

In LTKJAVA, Impinj uses the LTK-XML based construction of *ROSpec*. Figure 9.4 shows an excerpt of the *ROSpec* with the *C1G2Filters* added to the *C1G2InventoryCommand*.

LTK-XML

```
<AntennaConfiguration>
  <AntennaID>0</AntennaID>
  <C1G2InventoryCommand>
      <TagInventoryStateAware>false</TagInventoryStateAware>
      <C1G2Filter>
        <T>Do_Not_Truncate</T>
        <C1G2TagInventoryMask>
```

```
            <MB>1</MB>
            <Pointer>32</Pointer>
            <!-- pointer is in decimal -->
            <TagMask Count="8">33</TagMask>
        </C1G2TagInventoryMask>
        <C1G2TagInventoryStateUnawareFilterAction>
            <Action>Select_Unselect</Action>
        </C1G2TagInventoryStateUnawareFilterAction>
      </C1G2Filter>
      <C1G2Filter>
        <T>Do_Not_Truncate</T>
        <C1G2TagInventoryMask>
            <MB>1</MB>
            <Pointer>32</Pointer>
            <!-- pointer is in decimal -->
            <TagMask Count="8">35</TagMask>
        </C1G2TagInventoryMask>
        <C1G2TagInventoryStateUnawareFilterAction>
            <Action>Select_DoNothing</Action>
        </C1G2TagInventoryStateUnawareFilterAction>
      </C1G2Filter>
  </C1G2InventoryCommand>
</AntennaConfiguration>
```

Figure 9.4 LTKJAVA – InventoryFilters for docSample3

9.5 Tag Access

In this use case, we need an access spec to read 4 bytes from user memory on tags that are personnel tags, which correspond to GID in this example. The target tag *MB* (memory bank) is set to 1 corresponding to EPC memory. We want to *match* (run the access spec) on tags that start with hex 0x300035. This corresponds to the 16 PC (protocol control) bits in gen2 and in the EPCglobal EPC header that corresponds to a GID (0x35). Use the *TagMask* parameter to ignore the last 11 bits of the PC word, comparing only against the first 5 bits, which is the length. This ensures that we only get 96-bit GIDs that match this access spec. Once a tag matches, the Reader performs a read of the first two words of user memory (*MB* 3).

Note: This command has two *MB fields.* The first is in the target tag and determines which tags this *AccessSpec* applies to. The second is in the Read command, which performs the read.

Note: The pointers and lengths in the target tag parameter are specified in bits. The pointer and length are specified in 16-bit words.

> **Impinj Best Practice!** *Because the access filters offer the "don't care" mask, we can filter on the entire PC and EPC header and mark the 11 bits of the PC that can be variable as a "don't care". This adds extra protection against proprietary EPC formats colliding with the standard formats (GID and GRAI in this example).*
>
> **Impinj Best Practice!** *When using AccessSpecs, be careful with the AntennaID and ROSpecID. The AccessSpec will not run if these numbers are set incorrectly. We recommend setting these to zero if there is no specific coupling of the AccessSpec to either an antenna or ROSpec.*

9.5.1 LTKNET

In LTKNET, we demonstrate building *AccessSpecs* via LTK-XML and C# object creation. In an application, it is typical to choose either LTK-XML or C# object-based construction for each message type. Both methods are shown below, which provides a comparison so you can choose the best method for your application. Figure 9.5 shows the LTK-XML for the *AccessSpec* required for this use case. Figure 9.6 shows the C# code required to import and send this *AccessSpec* to the Reader. Figure 9.7 shows the same *AccessSpec* but with the use of C# objects to construct the various parameters and fields.

Note that the *TagMask* and *TagData*, which mark the 11 *C1G2PC* bits as "don't care".

LTK-XML

```xml
<?xml version="1.0" encoding="utf-8" ?>

<ADD_ACCESSSPEC
   xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:llrp="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.6"
   xsi:schemaLocation="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0
   http://www.llrp.org/ltk/schema/core/encoding/xml/1.0/llrp.xsd
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6
   http://developer.impinj.com/ltk/schema/encoding/xml/1.6/impinj.xsd"
   MessageID="0">
  <AccessSpec>
```

```
  <AccessSpecID>23</AccessSpecID>
  <AntennaID>0</AntennaID>
  <!-- 0 means to work on all antennas -->
  <ProtocolID>EPCGlobalClass1Gen2</ProtocolID>
  <CurrentState>Disabled</CurrentState>
  <ROSpecID>0</ROSpecID>
  <!--0 means to work with any RO Spec -->
  <AccessSpecStopTrigger>
     <AccessSpecStopTrigger>Null</AccessSpecStopTrigger>
     <OperationCountValue>0</OperationCountValue>
     <!--OperationCountValue is ignored since we are not using the trigger -->
  </AccessSpecStopTrigger>
  <AccessCommand>
     <C1G2TagSpec>
        <C1G2TargetTag>
           <MB>1</MB>
           <Match>true</Match>
           <Pointer>16</Pointer>
           <!--GID-96 looks like hex 300035 -->
           <!-- Use the mask so the 11 remaining PC bits are don't care -->
           <TagMask>f800ff</TagMask>
           <TagData>300035</TagData>
        </C1G2TargetTag>
     </C1G2TagSpec>
     <!--read the first two words of user memory-->
     <C1G2Read>
        <OpSpecID>1</OpSpecID>
        <AccessPassword>0</AccessPassword>
        <MB>3</MB>
        <WordPointer>0</WordPointer>
        <WordCount>2</WordCount>
     </C1G2Read>
  </AccessCommand>
  <AccessReportSpec>
     <!--Report when the access spec completes (e.g. read is done) -->
     <AccessReportTrigger>End_Of_AccessSpec</AccessReportTrigger>
  </AccessReportSpec>
  </AccessSpec>
</ADD_ACCESSSPEC>
```

Figure 9.5 LTK-XML –AccessSpec to read Tag User Memory

The C# code required to pass this command to the Reader is similar to the code in Section 9.4.1.

```
#region ADDAccessSpecWithXML
{
   Console.WriteLine("Adding AccessSpec from XML file \\n");
   Org.LLRP.LTK.LLRPV1.DataType.Message obj;
   ENUM_LLRP_MSG_TYPE msg_type;

   // read the XML from a file and validate its an ADD_ACCESS_SPEC
   try
   {
      FileStream fs = new FileStream(@"..\\..\\addAccessSpec.xml", FileMode.Open);
      StreamReader sr = new StreamReader(fs);
      string s = sr.ReadToEnd();
      fs.Close();
      LLRPXmlParser.ParseXMLToLLRPMessage(s, out obj, out msg_type);
      if (obj == null \\ msg_type != ENUM_LLRP_MSG_TYPE.ADD_ACCESSSPEC)
      {
         Console.WriteLine("Could not extract message from XML");
         reader.Close();
         return;
      }
   }
   catch
   {
      Console.WriteLine("Unable to convert to valid XML");
      reader.Close();
      return;
   }
   // Communicate that message to the reader
   MSG_ADD_ACCESSSPEC msg = (MSG_ADD_ACCESSSPEC)obj;
   MSG_ERROR_MESSAGE msg_err;
   MSG_ADD_ACCESSSPEC_RESPONSE rsp =

      reader.ADD_ACCESSSPEC(msg, out msg_err, 12000);
   if (rsp != null)
   {
      if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
      {
         Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
```

```
            reader.Close();
            return;
        }
    }
    else if (msg_err != null)
    {
        Console.WriteLine(msg_err.ToString());
        reader.Close();
        return;
    }
    else
    {
        Console.WriteLine("ADD_ACCESSSPEC Command Timed out\\n");
        reader.Close();
        return;
    }
}
#endregion
```

Figure 9.6 LTKNET –Adding an AccessSpec with LTK-XML

Adding the same *AccessSpec* (except for a unique ID) that uses C# programming objects rather than XML is shown in Figure 9.7.

LTKNET

```
{
    /* This section adds a second accessSpec identical to the
    first (except for its ID). This is duplicate code with
    the goal of showing an example of how to build LLRP specs
    from C# objects rather than XML */
    Console.WriteLine("Adding AccessSpec from C# objects \\n");

    /* create the target tag filter spec to perform access only on these tags
    This only requires a single filter (LTK/LLRP supports up to 2 ) */
    PARAM_C1G2TargetTag[] targetTag = new PARAM_C1G2TargetTag[1];
    targetTag[0] = new PARAM_C1G2TargetTag();
    targetTag[0].Match = true;
    targetTag[0].MB = new TwoBits(1);
    targetTag[0].Pointer = 16;
    targetTag[0].TagData = LLRPBitArray.FromHexString("300035");
```

```
targetTag[0].TagMask = LLRPBitArray.FromHexString("f800ff");
PARAM_C1G2TagSpec tagSpec = new PARAM_C1G2TagSpec();
tagSpec.C1G2TargetTag = targetTag;

// create the read operation to perform when this accessSpec is run
PARAM_C1G2Read read = new PARAM_C1G2Read();
read.AccessPassword = 0;
read.MB = new TwoBits(3);
read.WordCount = 2;
read.WordPointer = 0;
read.OpSpecID = 2;

// add the opSpec and the TagSpec to the AccessCmd
PARAM_AccessCommand accessCmd = new PARAM_AccessCommand();
accessCmd.AirProtocolTagSpec = new UNION_AirProtocolTagSpec();
accessCmd.AirProtocolTagSpec.Add(tagSpec);
accessCmd.AccessCommandOpSpec.Add(read);

// create the stop trigger for the Access Spec
PARAM_AccessSpecStopTrigger stop = new PARAM_AccessSpecStopTrigger();
stop.AccessSpecStopTrigger = ENUM_AccessSpecStopTriggerType.Null;
stop.OperationCountValue = 0;

// Create and set up the basic accessSpec
PARAM_AccessSpec accessSpec = new PARAM_AccessSpec();
accessSpec.AccessSpecID = 24;
accessSpec.AntennaID = 0;
accessSpec.ROSpecID = 0;
accessSpec.CurrentState = ENUM_AccessSpecState.Disabled;
accessSpec.ProtocolID = ENUM_AirProtocols.EPCGlobalClass1Gen2;

// add the access command and stop trigger to the accessSpec
accessSpec.AccessCommand = accessCmd;
accessSpec.AccessSpecStopTrigger = stop;

// Add the Access Spec to the ADD_ACCESSSPEC message
MSG_ADD_ACCESSSPEC addAccess = new MSG_ADD_ACCESSSPEC();
addAccess.MSG_ID = msgID++;
addAccess.AccessSpec = accessSpec;
```

```
    // communicate the message to the reader
    MSG_ERROR_MESSAGE msg_err;
    MSG_ADD_ACCESSSPEC_RESPONSE rsp =
       reader.ADD_ACCESSSPEC(addAccess, out msg_err, 12000);
    if (rsp != null)
    {
       if (rsp.LLRPStatus.StatusCode != ENUM_StatusCode.M_Success)
       {
          Console.WriteLine(rsp.LLRPStatus.StatusCode.ToString());
          reader.Close();
          return;
       }
    }
    else if (msg_err != null)
    {
       Console.WriteLine(msg_err.ToString());
       reader.Close();
       return;
    }
    else
    {
       Console.WriteLine("ADD_ACCESSSPEC Command Timed out\\n");
       reader.Close();
       return;
    }
}
#endregion
```

Figure 9.7 LTKNET –Adding an AccessSpec with C# Objects

Processing RO_ACCESS_REPORTS that contain *OpSpecResults* is similar to what we saw in Section 7. One notable difference is shown in Figure 9.8.

Note: With accumulation enabled on the reports, it is still possible to get multiple *AccessSpecs* for the same tag. This occurs if the Reader tries multiple times to access the same tag data and gets different results. Different results can occur because of errors reported by the Tag, or Reader.

LTKNET

```
#region CheckForAccessResults
// check for read data results
if ((msg.TagReportData[i].AccessCommandOpSpecResult != null))
```

```
{
   // there had better be one (since that what I asked for
   if (msg.TagReportData[i].AccessCommandOpSpecResult.Count == 1)
   {
      // it had better be the read result
      if (msg.TagReportData[i].AccessCommandOpSpecResult[0].GetType()
         == typeof(PARAM_C1G2ReadOpSpecResult))
      {
         PARAM_C1G2ReadOpSpecResult read = (PARAM_C1G2ReadOpSpecResult)
            msg.TagReportData[i].AccessCommandOpSpecResult[0];
         accessCount++;
         data += " AccessData: ";

         if (read.Result == ENUM_C1G2ReadResultType.Success)
         {
            data += read.ReadData.ToHexWordString();
         }
         else
         {
            data += read.Result.ToString();
         }
      }
   }
}
#endregion
```

Figure 9.8 LTKNET –Processing AccessSpec results in TagReportData

9.5.2 LTKCPP

We build the ACCESS_SPEC message much like other messages in LTKCPP. Note the *TagMask* and *TagData*, which mark the 11 *C1G2PC* bits as “don’t care”.

LTKCPP

```
{
   CADD_ACCESSSPEC * pCmd;
   CMessage * pRspMsg;
   CADD_ACCESSSPEC_RESPONSE * pRsp;
   pCmd = new CADD_ACCESSSPEC();
```

```
pCmd->setMessageID(m_messageID++);

/* build the C1G2Target Tag with the AccessSpec filter */
CC1G2TargetTag *ptargetTag = new CC1G2TargetTag();
ptargetTag->setMatch(true);
ptargetTag->setMB(1);
ptargetTag->setPointer(16);
llrp_u1v_t tagData = llrp_u1v_t(24);
tagData.m_nBit = 24;
tagData.m_pValue[0] = 0x30;
tagData.m_pValue[1] = 0x00;
tagData.m_pValue[2] = 0x35;
ptargetTag->setTagData(tagData);
llrp_u1v_t tagMask = llrp_u1v_t(24);
tagMask.m_nBit = 24;
tagMask.m_pValue[0] = 0xf8;
tagMask.m_pValue[1] = 0x00;
tagMask.m_pValue[2] = 0xff;
ptargetTag->setTagMask(tagMask);

/* build the AirProtocolTagSpec Add the filter */
CC1G2TagSpec *ptagSpec = new CC1G2TagSpec();
ptagSpec->addC1G2TargetTag(ptargetTag);

/* Build the read Op Spec */
CC1G2Read *pread = new CC1G2Read();
pread->setAccessPassword(0);
pread->setMB(3);
pread->setOpSpecID(1);
pread->setWordCount(2);
pread->setWordPointer(0);

/* Create the AccessCommand. Add the TagSpec and the OpSpec */
CAccessCommand *pAccessCommand = new CAccessCommand();
pAccessCommand->setAirProtocolTagSpec(ptagSpec);
pAccessCommand->addAccessCommandOpSpec(pread);

/* set up the Access Report Spec rule to report only with ROSpecs */
CAccessReportSpec *pAccessReportSpec = new CAccessReportSpec();
pAccessReportSpec->setAccessReportTrigger(
```

```
        AccessReportTriggerType_Whenever_ROReport_Is_Generated);

    /* set up the stop trigger for the access spec. Do not stop */
    CAccessSpecStopTrigger *pAccessStopTrigger = new CAccessSpecStopTrigger();
    pAccessStopTrigger->setAccessSpecStopTrigger(
    AccessSpecStopTriggerType_Null);
    pAccessStopTrigger->setOperationCountValue(0); /* ignored */

    /* Create and configure the AccessSpec */
    CAccessSpec *pAccessSpec = new CAccessSpec();
    pAccessSpec->setAccessSpecID(23);
    pAccessSpec->setAntennaID(0); /* valid for all antennas */
    pAccessSpec->setCurrentState(AccessSpecState_Disabled);
    pAccessSpec->setProtocolID(AirProtocols_EPCGlobalClass1Gen2);
    pAccessSpec->setROSpecID(0); /* valid for All RoSpecs */
    pAccessSpec->setAccessSpecStopTrigger(pAccessStopTrigger);
    pAccessSpec->setAccessReportSpec(pAccessReportSpec);
    pAccessSpec->setAccessCommand(pAccessCommand);
    /* Add the AccessSpec to the ADD_ACCESS_SPEC message */
    pCmd->setAccessSpec(pAccessSpec);

    /* Send the message, expect a certain type of response */
```

Figure 9.9 LTKCPP –Creating an AccessSpec

Processing RO_ACCESS_REPORTS that contain *OpSpecResult***s** is similar to what we saw in Section 7. One notable difference is shown in Figure 9.10.

Note: With accumulation enabled on the reports, it is still possible to get multiple *AccessSpecs* for the same tag. This occurs if the Reader tries multiple times to access the same tag data and gets different results. Different results can occur because of errors reported by the Tag, or Reader.

LTKCPP

```
{
    char aBuf[64];
    char bBuf[64];
    std::list<CParameter *>::iterator OpSpecResults;

    /*
    * Print the EPC. It could be an 96-bit EPC_96 parameter
    * or an variable length EPCData parameter.
```

```
    */
    CParameter * pEPCParameter = pTagReportData->getEPCParameter();
    formatOneEPC(pEPCParameter, aBuf, 64);

    /* This section only handles ReadResults. It can be extended in a
    similar fashion to handle all OpSpecResults */
    bBuf[0] = '\\0';
    for (
       OpSpecResults = pTagReportData->beginAccessCommandOpSpecResult();
       OpSpecResults != pTagReportData->endAccessCommandOpSpecResult();
       OpSpecResults++)
    {
       if( (*OpSpecResults)->m_pType ==
               &CC1G2ReadOpSpecResult::s_typeDescriptor)
       {
          formatOneReadResult(*OpSpecResults, bBuf, 64);
       }
    }
    /* End of line */
    printf("EPC: %s %s\\n", aBuf, bBuf);
}
```

Figure 9.10 LTKCPP –Processing AccessSpec results in TagReportData

9.5.3 LTKJAVA

Build the ACCESS_SPEC similar to other LTK-XML message in LTKJAVA. It is possible to construct the message with the objects, but this example uses LTK-XML code to demonstrate this process.

```
private void addAccessSpec() {
    LLRPMessage response;
    logger.info("Loading ADD_ACCESSSPEC message from file ADD_ACCESSSPEC.xml...");
    try {
       LLRPMessage addAccessMsg = Util.loadXMLLLRPMessage(new File( "./source/
           org/impinj/llrp/ltk/examples/docsample3/ADD_ACCESSSPEC.xml"));

       // TODO make sure this is an ADD_ACCESSSPEC message
       response = connection.transact(addAccessMsg, 10000);
```

```
      // check whether ADD_ACCESSSPEC addition was successful
      StatusCode status = ((ADD_ACCESSSPEC_RESPONSE)response).
          getLLRPStatus().getStatusCode();

      if (status.equals(new StatusCode("M_Success"))) {
         logger.info("ADD_ACCESSSPEC was successful");
         // save a copy for later
         accessspec = ((ADD_ACCESSSPEC) addAccessMsg).getAccessSpec();
      }
      else {
         logger.info(response.toXMLString());
         logger.info("ADD_ACCESSSPEC failures");
         System.exit(1);
      }

   } catch (TimeoutException ex) {
      logger.error("Timeout waiting for ADD_ACCESSSPEC response");
      System.exit(1);

   } catch (FileNotFoundException ex) {
      logger.error("Could not find file");
      System.exit(1);

   } catch (IOException ex) {
      logger.error("IO Exception on file");

   System.exit(1);
      } catch (JDOMException ex) {
      logger.error("Unable to convert LTK-XML to DOM");
      System.exit(1);
   } catch (InvalidLLRPMessageException ex) {
      logger.error("Unable to convert LTK-XML to Internal Object");
      System.exit(1);
   }
}
```

Figure 9.11 LTKJAVA –Creating an AccessSpec

Processing the results in the tag reports, we use the standard list iteration for the **opSpecResultList**. Figure 9.12 shows a snippet of code that only looks for the C1G2ReadResult that we requested above.

LTKJAVA

```
List<AccessCommandOpSpecResult> oplist = tr.getAccessCommandOpSpecResultList();
for (AccessCommandOpSpecResult opr : oplist) {
   if (opr.getClass() == C1G2ReadOpSpecResult.class) {
      epcString += " Read: ";
      C1G2ReadOpSpecResult rr = (C1G2ReadOpSpecResult) opr;
      epcString += " Status: " + rr.getResult().toString();
      epcString += " Data: " + rr.getReadData().toString();
   }
}
```

Figure 9.12 LTKJAVA –Processing AccessSpec results in TagReportData

> **Note:** With accumulation enabled on the reports it is possible to get multiple AccessSpecs for the same tag. This occurs if the Reader tries multiple times to access the same tag data and gets different results. Different results can occur because of errors reported by the Tag or Reader.

9.6 Report Generation

By use case analysis, we determined that the application polls the Reader for tag report data, and includes both inventory and access reports. LLRP uses the GET_REPORT message to trigger the Reader and to generate an RO_ACCESSS_REPORT message. It then clears the internal database and begins collecting new reports.

> **Impinj Best Practice!** *The GET_REPORT message should be thought of as a TRIGGER_REPORT. It is not a synchronous call. The Reader always generates an RO_ACCESS_REPORT for every GET_REPORT message, but the LTK treats the request and the corresponding reports as asynchronous messages.*

Polling the Reader involves periodically generating the GET_REPORT message and delivering it to the Reader. No change is required in the RO_ACCESS_REPORT handling, which was demonstrated in previous examples.

9.6.1 LTKNET

Polling for report data in LTKNET involves periodically building a simple GET_REPORT message and sending it to the Reader.

LTKNET

```
// wait around to collect some data.
for (int delay = 0; delay < 5; delay++)
{
   Thread.Sleep(30000);
   #region PollReaderReports
   {
      Console.WriteLine("Polling Report Data\\n");
      MSG_GET_REPORT msg = new MSG_GET_REPORT();
      MSG_ERROR_MESSAGE msg_err;
      msg.MSG_ID = msgID++;
      reader.GET_REPORT(msg, out msg_err, 10000);
   }
   #endregion

}
```

Figure 9.13 LTKNET –Trigger Asynchronous reports with GET_REPORT

9.6.2 LTKCPP

Polling for report data in LTKCPP involves periodically building a simple GET_REPORT message and sending it to the Reader.

> if(difftime(tempTime, pollTime) > 10) { /* poll the reader for its report data / *CGET_REPORT*preport = new CGET_REPORT(); sendMessage(preport); delete preport; pollTime = tempTime; }

Figure 9.14 LTKCPP –Trigger Asynchronous reports with GET_REPORT

9.6.3 LTKJAVA

Polling for report data in LTKJAVA involves periodically building a simple GET_REPORT message and sending it to the Reader. Note that this employs the ***send*** method rather than the ***transact*** method.

LTKJAVA

```java
private void getReport() {
   GET_REPORT grep = new GET_REPORT();
   grep.setMessageID(getUniqueMessageID());
   logger.info("Sending GET_REPORT ...");
   connection.send(grep);
}
```

Figure 9.15 LTKJAVA –Trigger Asynchronous reports with GET_REPORT

10 Impinj Octane LTK Tag Data

This section shows how to use the Impinj LTK to access the custom tag data elements in Octane Impinj LLRP. For a complete description of the Octane LLRP extensions, see the reference in Section 1.6.2. Impinj LTK includes this example as **docsample4**.

This example uses the SET_READER_CONFIG message to set the Reader's settings, including the Impinj settings for this use case.

> An application wants to get more low-level data about tags as they pass through a portal. The application will use this data to estimate the instantaneous velocity of the tag, to help determine its direction and timing through the portal.

10.1 Analysis

To use the low-level data available via Speedway, the application must configure the tag data that is reported through LLRP. Some of the data is available from the standard *TagReportContentSelector* parameter. The remaining Impinj-specific data is available through the *ImpinjTagReportContentSelector* parameter, which is a custom extension to the standard LLRP.

The use case requires multiple reads to make a good estimate of velocity, so the application enables "dual-target" inventory with Gen2 session 2. Setting the mode involves a tradeoff between the number of samples and the accuracy of these samples (see the same application node). In this case, DRM M=4 is chosen for the LTKCPP example, and Max throughput is chosen for the LTKNET example.

After the Reader is configured and low-level data is enabled on the Reader, the application must extract the data from the *RO_ACCESS_REPORT* message. Then the application must calculate the radial velocity of the tag. For a description of estimating velocity, see the reference in Section 1.6.8.

10.2 Application Flow

The steps required to implement this use case are:

1. Initialize the Library.
2. Connect to the Reader.
3. Enable Impinj Extensions.
4. Set the factory default LLRP configuration to ensure that the Reader is in a known state. (We rely on the default Reader configuration for this simple example.)

5. SET_READER_CONFIG with the appropriate low-level data enabled.
6. ADD_ROSPEC to tell the Reader to perform an inventory.
7. Enable ENABLE_ROSPEC.
8. START_ROSPEC to start the inventory operation.
9. Process RFID Data with low-level data to estimate velocity.

You have sample code for steps 1-4, and 6-8, so this discussion and example begins with step 5. For the complete sample code, see the **docSample4** directories in the Impinj LTK distribution.

Examples for both LTKCPP and LTKNET show the object-based parameter and message construction in the next section.

10.3 Impinj Configuration

Perform the application setup with the SET_READER_CONFIG message. This example uses the object-based construction with LTKCPP and LTK-XML based construction, using LTKNET to apply the following configuration.

10.3.1 LTK-XML

The following LTK-XML describes the configuration required for the application. Specifically, take note of the **TagReportContentSelector** and the *ImpinjTagReportContentSelector*, which enable the required data for our velocity estimate. Note how to set the *session*, *ImpinjInventorySearchMode*, and *ModeIndex* values in the code below.

LTK-XML

```
<?xml version="1.0" encoding="utf-8" ?>
   <SET_READER_CONFIG
      xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
      xmlns:llrp="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.0"
      xsi:schemaLocation="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0
      http://www.llrp.org/ltk/schema/core/encoding/xml/1.0/llrp.xsd
      http://developer.impinj.com/ltk/schema/encoding/xml/1.8
      http://developer.impinj.com/ltk/schema/encoding/xml/1.8/impinj.xsd"
      MessageID="0">
          <ResetToFactoryDefault>false</ResetToFactoryDefault>
          <AntennaConfiguration>
              <AntennaID>0</AntennaID>
```

```
          <C1G2InventoryCommand>
            <TagInventoryStateAware>false</TagInventoryStateAware>
            <C1G2RFControl>
                <ModeIndex>0</ModeIndex>
                <Tari>0</Tari>
            </C1G2RFControl>
            <C1G2SingulationControl>
                <!--Need Dual-target for lots of reads, so use session 2 -->
                <Session>2</Session>
                <TagPopulation>32</TagPopulation>
                <TagTransitTime>0</TagTransitTime>
            </C1G2SingulationControl>
            <ImpinjInventorySearchMode>
                xmlns="http://developer.impinj.com/ltk/schema/encoding/
                  xml/1.8"&gt;
                <!--Need Dual-target for lots of low level data -->
                <InventorySearchMode>Dual_Target</InventorySearchMode>
            </ImpinjInventorySearchMode>
            <ImpinjLowDutyCycle xmlns="http://developer.impinj.com/ltk/
                schema/encoding/xml/1.8">
                <LowDutyCycleMode>Disabled</LowDutyCycleMode>
                <EmptyFieldTimeout>10000</EmptyFieldTimeout>
                <FieldPingInterval>200</FieldPingInterval>
            </ImpinjLowDutyCycle>
          </C1G2InventoryCommand>
      </AntennaConfiguration>
      <ROReportSpec>
          <ROReportTrigger>Upon_N_Tags_Or_End_Of_ROSpec</ROReportTrigger>
          <N>1</N>
          <!--Want to report every tag for velocity estimate -->
          <TagReportContentSelector>
            <!-- Enable certain things that we'll use with low level data
                  especially antenna, channel, timestamp -->
            <EnableROSpecID>false</EnableROSpecID>
            <EnableSpecIndex>false</EnableSpecIndex>
            <EnableInventoryParameterSpecID>false
                </EnableInventoryParameterSpecID>
            <EnableAntennaID>true</EnableAntennaID>
            <EnableChannelIndex>true</EnableChannelIndex>
            <EnablePeakRSSI>true</EnablePeakRSSI>
```

```
            <EnableFirstSeenTimestamp>true</EnableFirstSeenTimestamp>
            <EnableLastSeenTimestamp>false</EnableLastSeenTimestamp>
            <EnableTagSeenCount>false</EnableTagSeenCount>
            <EnableAccessSpecID>false</EnableAccessSpecID>
            <C1G2EPCMemorySelector>
                <EnableCRC>false</EnableCRC>
                <EnablePCBits>false</EnablePCBits>
            </C1G2EPCMemorySelector>
          </TagReportContentSelector>
          <Impinj:ImpinjTagReportContentSelector xmlns=
              "http://developer.impinj.com/ltk/schema/encoding/xml/1.8">
            <ImpinjEnableSerializedTID><SerializedTIDMode>Disabled
                </SerializedTIDMode>
            </ImpinjEnableSerializedTID>
            <ImpinjEnableRFPhaseAngle>
                <RFPhaseAngleMode>Enabled</RFPhaseAngleMode>
            </ImpinjEnableRFPhaseAngle>
            <ImpinjEnablePeakRSSI>
                <PeakRSSIMode>Enabled</PeakRSSIMode>
            </ImpinjEnablePeakRSSI>
          </Impinj:ImpinjTagReportContentSelector>
      </ROReportSpec>
  </SET_READER_CONFIG>
```

Figure 10.1 LTK-XML –Setting Tag Report Contents

10.3.2 LTKNET

Using XML to set the configuration in LTKNET is identical to the previous examples.

10.3.3 LTKCPP

In LTKCPP, we use object-based construction. However, LTK-XML construction is also possible using the LTKCPP LTK-XML examples shown above. Previous examples show the construction based on SET_READER_CONFIG in LTKCPP. The example in Figure 10.2 provides just the section of code that adds the **CTagReportContentSelector**. See the file **docsample4.cpp** for the remaining code. Remember to add the tag report content selectors to the correct places in the **CROReportSpec.**

LTKCPP

```
/* lets turn off off report data that we don't need since our use
   case suggests we are bandwidth constrained */
CTagReportContentSelector *pROcontent = new CTagReportContentSelector();
pROcontent->setEnableAccessSpecID(false);

/* these are very handy to have with low level data */
pROcontent->setEnableAntennaID(true);
pROcontent->setEnableChannelIndex(true);
pROcontent->setEnableFirstSeenTimestamp(true);
pROcontent->setEnableInventoryParameterSpecID(false);
pROcontent->setEnableLastSeenTimestamp(false);
pROcontent->setEnablePeakRSSI(false);
pROcontent->setEnableROSpecID(false);
pROcontent->setEnableSpecIndex(false);
pROcontent->setEnableTagSeenCount(false);
CC1G2EPCMemorySelector *pC1G2Mem = new CC1G2EPCMemorySelector();
pC1G2Mem->setEnableCRC(false);
pC1G2Mem->setEnablePCBits(false);
pROcontent->addAirProtocolEPCMemorySelector(pC1G2Mem);
pROrs->setTagReportContentSelector(pROcontent);

/* Turn on the low level phase data in the ImpinjTagContentSelector*/
CImpinjTagReportContentSelector * pImpTagCnt = new
   CImpinjTagReportContentSelector();
CImpinjEnableRFPhaseAngle * pEnableRfPhase = new
   CImpinjEnableRFPhaseAngle();
pEnableRfPhase->setRFPhaseAngleMode(ImpinjRFPhaseAngleMode_Enabled);
pImpTagCnt->setImpinjEnableRFPhaseAngle(pEnableRfPhase);
CImpinjEnablePeakRSSI * pEnablePeakRssi = new CImpinjEnablePeakRSSI();
pEnablePeakRssi->setPeakRSSIMode(ImpinjPeakRSSIMode_Enabled);
pImpTagCnt->setImpinjEnablePeakRSSI(pEnablePeakRssi);
CImpinjEnableSerializedTID * pEnableSerializedTID = new
   CImpinjEnableSerializedTID();
pEnableSerializedTID->setSerializedTIDMode(ImpinjSerializedTIDMode_Disabled);
pImpTagCnt->setImpinjEnableSerializedTID(pEnableSerializedTID);
pROrs->addCustom(pImpTagCnt);
```

Figure 10.2 LTKCPP – Setting Tag Report Contents

10.3.4 LTKJAVA

Using XML to set the configuration in LTKJAVA is identical to the code in previous examples.

10.4 Getting Tag Low-Level Data

Low level tag information is contained within the LLRP *TagReportData as custom* parameters.

10.4.1 LTKCPP

Figure 10.3 shows how to get an individual tag report from the RO_ACCESS_REPORT message. After this report is extracted, the application must examine the report to find the low-level data.

In the LTKCPP, custom parameters are stored as STL lists. For tag information that was set through the *ImpinjTagReportContentSelector*, the data is reported via the *CustomParameter* within the standard LLRP *TagReportDataParameter*. Note that the type of each custom parameter is checked before the function is called to extract the relevant data.

LTKCPP

```
for(
   Cur = pTagReportData->beginCustom();
   Cur != pTagReportData->endCustom();
   Cur++)
{
   /* look for our special low-level data */
   if(&CImpinjRFPhaseAngle::s_typeDescriptor == (*Cur)->m_pType)
   {
      if(getOnePhaseAngle((CImpinjRFPhaseAngle*) *Cur, &phase))
      {
         written = snprintf(ptr, len, " ph=%+04d", (int) phase);
         ptr += written;
         len -= written;
      }
   }
}
```

Figure 10.3 LTKCPP – Extracting Impinj Tag Report Data

10.4.2 LTKNET

LTKNET stores tag report data information inside a custom parameter array. The application should validate the type of the parameter before casting it back to a *PARAM_ImpinjRFPhaseAngle*.

LTKNET

```
if (msg.TagReportData[i].Custom != null)
{
   for (int x = 0; x < msg.TagReportData[i].Custom.Length; x++)
   {
      // is it the rfphase report that we asked for
      if (msg.TagReportData[i].Custom[x].GetType() ==
            typeof(PARAM_ImpinjRFPhaseAngle))
      {
         PARAM_ImpinjRFPhaseAngle rfPhase =
         (PARAM_ImpinjRFPhaseAngle)msg.TagReportData[i].Custom[x];
         currentRfPhase = rfPhase.PhaseAngle;
         data += " Phase: " + currentRfPhase.ToString();
      }
   }
}
```

Figure 10.4 LTKNET – Extracting Impinj Tag Report Data

10.4.3 LTKJAVA

LTKJAVA stores tag report data inside a *custom* Parameter list. The application should validate the type of the parameter before casting it back from type **custom**. The code below shows a snippet from the tag data processing function.

LTKJAVA

```
List<Custom> clist = tr.getCustomList();
for (Custom cd : clist) {
   if (cd.getClass() == ImpinjRFPhaseAngle.class) {
      epcString += " ImpinjPhase: " +
      ((ImpinjRFPhaseAngle) cd).getPhaseAngle().toString();
   }
   if(cd.getClass() == ImpinjPeakRSSI.class) {
      epcString += " ImpinjPeakRSSI: " +
```

```
        ((ImpinjPeakRSSI) cd).getRSSI().toString();
    }
}
```

10.5 Computing Velocity

To compute the radial velocity of the tag from the report data, use the formulas and method outlined in the application note, which is referenced in Section 1.6.8. In addition, see the source code samples in **docsample4.cpp**, **docsample4.cs**, and **docsample4.java**.

11 Using Monza 4 QT

Monza 4 QT features allow control of public and private areas of the tag memory. It uses the same *AccessSpec* model as shown in Section 9. For a description of the Monza 4 QT feature set, see the reference in Section 1.6. The only new programming exercise to use Monza 4, aside from understanding the functionality, utilizes custom *AccessSpecs* via the *CustomParameter* in the LLRP standard, as discussed in this section.

11.1 Adding Custom OpSpecs

Add Custom OpSpecs to the standard list of OpSpecs by using the complete use case example, as shown in Section 9.

11.1.1 LTKCPP

LTKCPPuses the ***addAccessCommandOpSpec*** method to add a custom standard OpSpec to the **AccessCommand** object, as shown in Figure 11.1.

LTKCPP

```
/* Create the AccessCommand. Add the tag spec */
CAccessCommand *pAccessCommand = new CAccessCommand();
pAccessCommand->setAirProtocolTagSpec(ptagSpec);
/* Buld a new SetQT OpSpec */
{
   CImpinjSetQTConfig *psetQT = new CImpinjSetQTConfig();
   psetQT->setAccessPassword(m_password);
   psetQT->setOpSpecID(6);
   psetQT->setAccessRange(m_shortRange);
   psetQT->setDataProfile(ImpinjQTDataProfile_Public);
   psetQT->setPersistence(ImpinjQTPersistence_Permanent);
   pAccessCommand->addAccessCommandOpSpec(psetQT);
}
/* Add a set QT op Spec */
pAccessCommand->addAccessCommandOpSpec(psetQT);
```

Figure 11.1 LTKCPP – Adding Custom OpSpec

11.1.2 LTKNET

LTKNET uses the ***Add*** method of the **AccessCommandOpSpec** class to add a custom OpSpec. as shown in Figure 11.2.

LTKNET

```
PARAM_AccessCommand accessCmd = new PARAM_AccessCommand();
accessCmd.AirProtocolTagSpec = new UNION_AirProtocolTagSpec();
accessCmd.AirProtocolTagSpec.Add(tagSpec);
PARAM_ImpinjSetQTConfig setQTPrivate = new PARAM_ImpinjSetQTConfig();
setQTPrivate.OpSpecID = 7;
setQTPrivate.AccessPassword = m_password;
setQTPrivate.Persistence = ENUM_ImpinjQTPersistence.Permanent;
setQTPrivate.DataProfile = ENUM_ImpinjQTDataProfile.Private;
setQTPrivate.AccessRange = m_shortRange;
accessCmd.AccessCommandOpSpec.Add(setQTPrivate);
```

Figure 11.2 LTKNET – Adding Custom OpSpec

11.2 Retrieving Custom OpSpecResults

11.2.1 LTKCPP

In LTKCPP, *OpSpecResults* are stored as STL lists. The application must iterate the list and examine each type of result, and then act appropriately, as shown in Figure 11.3.

LTKCPP

```
for (
      OpSpecResults = pTagReportData->beginAccessCommandOpSpecResult();
      OpSpecResults != pTagReportData->endAccessCommandOpSpecResult();
      OpSpecResults++)
{
   if( (*OpSpecResults)->m_pType ==
        &CImpinjSetQTConfigOpSpecResult::s_typeDescriptor)
   {
      written = formatOneSetQTConfigResult(*OpSpecResults,
      ptr, bufLimit, "\\n SETQT ");
      ptr += written;
      bufLimit-=written;
```

```
    }
}
```

Figure 11.3 LTKCPP – Retreiving Custom OpSpecResults

11.2.2 LTKNET

In LTKNET, the application must loop through the list of *OpSpecResults* and compare the type of each one with the possible types available, as shown in Figure 11.4.

LTKNET

```
if ((msg.TagReportData[i].AccessCommandOpSpecResult != null))
{
    for(int x = 0; x < msg.TagReportData[i].AccessCommandOpSpecResult.Count; x++)
    {
        if (msg.TagReportData[i].AccessCommandOpSpecResult[x].GetType() ==
            typeof(PARAM_ImpinjSetQTConfigOpSpecResult))
        {
            PARAM_ImpinjSetQTConfigOpSpecResult set =
                (PARAM_ImpinjSetQTConfigOpSpecResult)msg.
                    TagReportData[i].AccessCommandOpSpecResult[x];
            opSpecCount++;
            data += "\\n setQTResult(" +
                set.OpSpecID.ToString() + "): " +
            set.Result.ToString();
        }
    }
}
```

Figure 11.4 LTKNET – Retrieving Custom OpSpecResults

12 xArray Gateway

xArray uses the LTK in a way similar to Speedway. The examples below show how to create an **ROSPEC** and **READER_CONFIGURATION** for the Location role in XML format, as well as examples of how to write event handlers in C# and Java.

Impinj Best Practice!

- *For optimal performance when using Location role: Use Mode index 1002 over 1000 on FCC and GX1 xArrays. Use Mode index 0 on ETSI and Japan xArrays.*
- *Get the Tilt configuration before running an RO_SPEC. If you get the Tilt configuration while an RO_SPEC is running, the value returned will be the value before the RO_SPEC started.*
- *You should set the Beacon state before starting the RO_SPEC. The Beacon state will not update until an RO_SPEC stops running.*

12.1 Example of ROSPEC for Location Role

Figure 12.1 provides an example of how to create a basic ROSPEC for the xArray location role in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<ADD_ROSPEC MessageID="0"
    xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.20"
    xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0">
<ROSpec>
    <ROSpecID>1</ROSpecID>
    <Priority>0</Priority>
    <CurrentState>Disabled</CurrentState>
    <ROBoundarySpec>
        <ROSpecStartTrigger>
           <ROSpecStartTriggerType>Null</ROSpecStartTriggerType>
        </ROSpecStartTrigger>
        <ROSpecStopTrigger>
            <ROSpecStopTriggerType>Null</ROSpecStopTriggerType>
            <DurationTriggerValue>0</DurationTriggerValue>
        </ROSpecStopTrigger>
    </ROBoundarySpec>
        <Impinj:ImpinjLISpec>
```

```
            <ImpinjLocationReporting>
                <EnableUpdateReport>true</EnableUpdateReport>
                <EnableEntryReport>true</EnableEntryReport>
                <EnableExitReport>true</EnableExitReport>
                <EnableDiagnosticReport>true</EnableDiagnosticReport>
            </ImpinjLocationReporting>
        </Impinj:ImpinjLISpec>
    </ROSpec>
</ADD_ROSPEC>
```

Figure 12.1 LTK-XML - Basic ROSPEC XML example

12.2 Create a READER_CONFIGURATION for xArray Location Role in XML

Figure 12.2 provides an example of how to create a basic READER_CONFIGURATION in XML for the xArray location role.

```
<?xml version="1.0"?>
<SET_READER_CONFIG MessageID="0"
       xmlns:Impinj="http://developer.impinj.com/ltk/schema/encoding/xml/1.20"
       xmlns="http://www.llrp.org/ltk/schema/core/encoding/xml/1.0">
    <ResetToFactoryDefault>false</ResetToFactoryDefault>
    <ImpinjLocationConfig xmlns=
                "http://developer.impinj.com/ltk/schema/encoding/xml/1.20">
        <UpdateIntervalSeconds>1</UpdateIntervalSeconds>
        <ComputeWindowSeconds>255</ComputeWindowSeconds>
        <TagAgeIntervalSeconds>300</TagAgeIntervalSeconds>
    </ImpinjLocationConfig>
    <ImpinjPlacementConfiguration
        xmlns="http://developer.impinj.com/ltk/schema/encoding/xml/1.20">
        <HeightCm>400</HeightCm>
        <FacilityXLocationCm>0</FacilityXLocationCm>
        <FacilityYLocationCm>0</FacilityYLocationCm>
        <OrientationDegrees>0</OrientationDegrees>
    </ImpinjPlacementConfiguration>
    <ImpinjC1G2LocationConfig
              xmlns="http://developer.impinj.com/ltk/schema/encoding/xml/1.20">
         <ModeIndex>0</ModeIndex>
```

```
        <Session>2</Session>
    </ImpinjC1G2LocationConfig>
</SET_READER_CONFIG>
```

Figure 12.2 LTK-XML - Basic READER_CONFIG XML example

12.3 Create an Event Handler for xArray in Java and C#

The following examples show how to write event handlers for xArray in C# and Java.

Figure 12.3 shows the event handler code that supports the location role in Java for xArray.

LTKJAVA

```
public void messageReceived(LLRPMessage message) {
    // log("Received " + message.getName() + " message asychronously");
    if (message.getTypeNum() == RO_ACCESS_REPORT.TYPENUM) {
        RO_ACCESS_REPORT report = (RO_ACCESS_REPORT) message;
        try {
            for (TagReportData tr : report.getTagReportDataList()) {
                printTagReport(tr);
            }
        } catch (Exception ex){
            System.out.println("printTagReport Exception");
        }

        try {
            for (Custom cust : report.getCustomList()) {
                checkVendor(cust);
                if (cust instanceof ImpinjExtendedTagInformation) {
                    ImpinjExtendedTagInformation tagInfo =
                        (ImpinjExtendedTagInformation)cust;
                    if (tagInfo.getImpinjLocationReportData() != null) {
                        System.out.println(impinjLocationOutput(tagInfo));
                    }
                }
            }

        } catch (Exception ex) {
            ex.printStackTrace();
```

```
        }
    } // RO_ACCESS_REPORT
}
// Build String with xArray Location report details.
private String impinjLocationOutput(ImpinjExtendedTagInformation tagInfo) {
    // Get the EPC associated with this location report
    // xArray reports one tag per report for location, so there is only one EPC
    String output = "Location EPCs: ";
    for (EPCData epc : tagInfo.getEPCDataList())
        output += epc.getEPC() + " ";
    // Set the confidence factors
    output += " ReadCnt=" + tagInfo.getImpinjLocationReportData().
       getImpinjLocationConfidence().getReadCount();
    output += " Confs:";
    for (int dataIndex = 0; dataIndex <
        tagInfo.getImpinjLocationReportData().getImpinjLocationConfidence().
          getConfidenceData().size(); dataIndex++)
        output += tagInfo.getImpinjLocationReportData().
            getImpinjLocationConfidence().getConfidenceData().get(dataIndex)
             + " ";
    // Get the other location report parameters
    output += " Xcm=" + tagInfo.getImpinjLocationReportData().
        getLocXCentimeters();
    output += " Ycm=" + tagInfo.getImpinjLocationReportData().
        getLocYCentimeters();
    output += " Type=" + tagInfo.getImpinjLocationReportData().getType();
    output += " TS=" + getDateStringFromEpoch(
       tagInfo.getImpinjLocationReportData().getTimestampUTC());
    return output;
}
```

Figure 12.3 LTKJAVA - Java xArray event handler example

Figure 12.4 shows the event handler code that supports the location role in C# for xArray.

LTKNET

```
public override void reader_OnRoAccessReportReceived(MSG_RO_ACCESS_REPORT msg)
{
  if (msg.Custom != null)
  {
```

```
      int numCustom = msg.Custom.Length;
      for (int i = 0; i < numCustom; i++)
      {
          if (msg.Custom[i] is PARAM_ImpinjExtendedTagInformation)
          {
              PARAM_ImpinjExtendedTagInformation tagInfo =
                  (PARAM_ImpinjExtendedTagInformation)msg.Custom[i];
              // We received a location report
              if (tagInfo.ImpinjLocationReportData != null)
              {
                  Console.WriteLine(ImpinjLocationOutput(tagInfo));
              }
          }
      }
  }
}

// Print out xArray Location report details.
private String ImpinjLocationOutput(PARAM_ImpinjExtendedTagInformation tagInfo)
{
    // Get the EPC associated with this location report
    // xArray reports one tag per report for location, so there is only one EPC
    string output = "Location EPCs: ";
    foreach (PARAM_EPCData epc in tagInfo.EPCData)
    {
        output += epc.EPC.ToHexString() + " ";
    }

    // Set the confidence factors
    output += "ReadCnt=" + tagInfo.ImpinjLocationReportData.
        ImpinjLocationConfidence.ReadCount;
    output += " Conf: ";
    for (int dataIndex = 0; dataIndex < tagInfo.ImpinjLocationReportData.
            ImpinjLocationConfidence.ConfidenceData.Count; dataIndex++)
    {
        output += tagInfo.ImpinjLocationReportData.ImpinjLocationConfidence.
           ConfidenceData[dataIndex] + " ";
    }

    // Get the other location report parameters
```

```
    output += " Xcm=" + tagInfo.ImpinjLocationReportData.LocXCentimeters;
    output += " Ycm=" + tagInfo.ImpinjLocationReportData.LocYCentimeters;
    output += " Type=" + tagInfo.ImpinjLocationReportData.Type;
    output += " TS=" + FromEpochTime(tagInfo.ImpinjLocationReportData.
        TimestampUTC).ToLocalTime();
    return output;
}
```

Figure 12.4 LTKNET - C# xArray event handler example

13 Revision History

Date	Revision	Comments
09/01/2009	10.8.0.240	Initial release
03/14/2010	10.10.0.240	Update for addition of LTK-XML to LTKCPP
10/28/2010	10.12.0.240	Update for Octane 4.6
4/25/2011	10.14.0.240	Add support for LTKJAVA
4/30/2012	10.16.0.240	Update for Octane 4.10, no changes
12/16/2014	10.18.1.240	Update for Octane 5.2
10/10/2016	10.20.1.240	Update for Octane 5.8.0 - no changes
12/07/2020	10.20.1.240	Update for Octane 7.4.0 - formatting code

14 Notices

Copyright © 2021, Impinj, Inc. All rights reserved.

Impinj gives no representation or warranty, express or implied, for accuracy or reliability of information in this document. Impinj reserves the right to change its products and services and this information at any time without notice.

EXCEPT AS PROVIDED IN IMPINJ'S TERMS AND CONDITIONS OF SALE (OR AS OTHERWISE AGREED IN A VALID WRITTEN INDIVIDUAL AGREEMENT WITH IMPINJ), IMPINJ ASSUMES NO LIABILITY WHATSOEVER AND IMPINJ DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATED TO SALE AND/OR USE OF IMPINJ PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT.

NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY PATENT, COPYRIGHT, MASK WORK RIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT IS GRANTED BY THIS DOCUMENT.

Impinj assumes no liability for applications assistance or customer product design. Customers should provide adequate design and operating safeguards to minimize risks.

Impinj products are not designed, warranted or authorized for use in any product or application where a malfunction may reasonably be expected to cause personal injury or death or property or environmental damage ("hazardous uses") or for use in automotive environments. Customers must indemnify Impinj against any damages arising out of the use of Impinj products in any hazardous or automotive uses.

Impinj, GrandPrix ™, Indy ®, Monza ®, Octane ™, QT ®, Speedway ®, STP ™, True3D ™, xArray ®, and xSpan ® are trademarks or registered trademarks of Impinj, Inc. All other product or service names are trademarks of their respective companies.

These products may be covered by one or more U.S. patents. See http://www.impinj.com/patents for details.

For more information, contact support@impinj.com