UHF Gen 2 RFID Speedway® Revolution

Impinj LTK Programmers' Guide

REV 10.14.0 25-April-2011

Proprietary and Confidential

www.impinj.com

Copyright © 2009, 2010, 2011 Impinj, Inc.
Impinj, Octane, and Speedway are either
registered trademarks or trademarks of Impinj, Inc.

For more information, contact rfid_info@impinj.com

Table of Contents

1 Introduction 9
1.1 Purpose of Guide 9
1.2 Intended Audience 9
1.3 Overview of Contents 9
1.4 Document Conventions and Terms 9
1.4.1 Document Terms 11
1.5 Technical Support 11
1.6 Other Reference Material 11
1.6.1 LLRP Standards Document 11
1.6.2 Octane LLRP Reference Guide 12
1.6.3 Speedway Embedded Developers' Guide 12
1.6.4 Language-Specific API Reference Documentation 12
1.6.5 www.LLRP.org 13
1.6.6 LLRP Toolkit Project on Source-Forge 13
1.6.7 Impinj LTK Extension Files 13
1.6.8 Impinj Low Level User Data Support Application Note 13
2 LLRP Overview 13
2.1 Origin and Purpose 13
2.2 Connection Details 14
2.3 Declarative Specifications 15
2.4 Messages, Parameters, and Fields 15
2.4.1 Messages 15
2.4.2 Fields 17
2.4.3 Parameters 18
2.5 Introduction to LLRP Specs 18
2.5.1 Reader Operation Specs (ROSpecs) 18
2.5.2 Access Specs 21
2.6 A typical LLRP Exchange 22
2.7 Vendor Extensions to LLRP 23
2.8 Impinj LLRP Extensions 23
3 LTK Overview 24
3.1 Origin and Purpose 24
3.2 LTK versus LLRP 24
3.3 LTK Definition Files 24
3.4 LTK Vendor Extensions 26
3.5 LTK-XML 27
4 Getting Started with the Impinj LTK 29
4.1 Overview 29

Copyright © 2009, 2010, 2011 Impinj, Inc.

4.1.1 Impinj Supported Platforms and Languages 29
4.1.2 CHANGES.TXT 29
4.1.3 Versioning 30
4.2 Obtaining the LTK 30
4.3 LTK Required Tools/Environment 30
4.4 Installing and Running the LTK Samples 30
4.4.1 LTKC 30
4.4.2 LTKCPP 31
4.4.3 LTKNET 33
4.4.4 LTKJava 36
4.4.4.1 Windows 36
4.4.4.2 Linux 39
5 LTK Language Details 41
5.1 LTKCPP (LTKC) 41
5.1.1 Connection Class 41
5.1.2 Element Class 42
5.1.3 Fields, Subparameters, and Accessor Functions 43
5.1.4 Why It Is Best to Use the Accessor Functions 44
5.1.5 How to Determine the Type of an Element 45
5.1.6 Practical Information on Building Message in LTKCPP 45
5.1.7 Error Reporting (ErrorDetails Class) 47
5.1.8 Extensions 48
5.1.8.1 LLRP Model of Extensions 48
5.1.8.2 Constructing Custom Elements 48
5.1.8.3 Accessors 48
5.1.8.4 Enrollment in TypeRegistry 49
5.2 LTKNET 49
5.2.1 LLRPClient 49
5.2.2 Impinj Installer Class 50
5.2.3 LLRP Message Classes 50
5.2.4 Handling ROSpecs 51
5.2.5 Handling AccessSpecs 52
5.3 LTKJAVA 53
5.3.1 Receiving LLRP Messages 54
5.3.2 Connections 54
5.3.3 Sending and Receiving Messages 55
5.3.4 LLRPMessage and LLRPParameter 55
5.3.5 Building LLRP Messages 55
5.3.6 Custom Extensions 56
5.3.7 Logging 57
5.3.8 Summary 57

Copyright © 2009, 2010, 2011 Impinj, Inc.

6 Adding the LTK into your programs ... 57
6.1 Compiling and Linking ... 57
6.2 Initializing the library ... 60
6.3 Connecting to the Reader ... 61
6.3.1 LTKCPP ... 62
6.3.2 LTKNET ... 64
6.3.3 LTKJAVA ... 65
6.4 Enabling the Impinj Extensions ... 65
6.4.1 LTKCPP ... 65
6.4.2 LTKNET ... 67
6.4.3 LTKJAVA ... 67
6.5 Debugging with LTK ... 68
6.5.1 LTKC ... 68
6.5.2 LTKCPP ... 69
6.5.3 LTKNET ... 69
6.5.4 LTKJava ... 70
7 Creating a Simple Impinj LTK Application ... 70
7.1 Application Flow ... 70
7.2 Setup ... 72
7.3 Factory Default the Reader ... 72
7.3.1 LTKCPP ... 72
7.3.2 LTKNET ... 74
7.3.3 LTKJAVA ... 74
7.4 Add an ROSpec ... 75
7.4.1 LTKCPP ... 76
7.4.2 LTKNET ... 79
7.4.3 LTKJava ... 81
7.5 Enable an ROSpec ... 82
7.6 Start an ROSpec ... 85
7.7 Handling Tag Report Data ... 88
7.7.1 LTKCPP ... 88
7.7.2 LTKNET ... 89
7.7.3 LTKJava ... 91
8 Impinj Octane LTK Application ... 93
8.1 Analysis ... 94
8.2 Application Flow ... 94
8.3 Impinj Capabilities/Getting Reader Capabilities ... 95
8.4 Get Reader Configuration ... 100
8.5 Impinj Configuration ... 104
8.5.1 LTK-XML ... 104
8.5.2 LTKNET ... 105

Copyright © 2009, 2010, 2011 Impinj, Inc.

8.5.3 LTKCPP ... 107
8.5.4 LTKJAVA ... 109
9 LLRP Access, Filtering, and Accumulation ... 110
9.1 Analysis ... 111
9.2 Application Flow ... 111
9.3 Report Accumulation ... 111
9.3.1 LTKNET ... 112
9.3.2 LTKCPP ... 113
9.3.3 LTKJAVA ... 114
9.4 Tag Filtering ... 114
9.4.1 LTKNET ... 115
9.4.2 LTKCPP ... 115
9.4.3 LTKJAVA ... 117
9.5 Tag Access ... 117
9.5.1 LTKNET ... 118
9.5.2 LTKCPP ... 123
9.5.3 LTKJAVA ... 125
9.6 Report Generation ... 126
9.6.1 LTKNET ... 127
9.6.2 LTKCPP ... 127
9.6.3 LTKJAVA ... 127
10 Impinj Octane LTK Tag Data ... 128
10.1 Analysis ... 128
10.2 Application Flow ... 128
10.3 Impinj Configuration ... 129
10.3.1 LTK-XML ... 129
10.3.2 LTKNET ... 130
10.3.3 LTKCPP ... 130
10.3.4 LTKJAVA ... 131
10.4 Getting Tag Low-Level Data ... 131
10.4.1 LTKCPP ... 131
10.4.2 LTKNET ... 132
10.4.3 LTKJAVA ... 132
10.5 Computing Velocity ... 133
11 Using Monza4 QT ... 133
11.1 Adding Custom OpSpecs ... 133
11.1.1 LTKCPP ... 133
11.1.2 LTKNET ... 134
11.2 Retrieving Custom OpSpecResults ... 134
11.2.1 LTKCPP ... 134
11.2.2 LTKNET ... 134

Copyright © 2009, 2010, 2011 Impinj, Inc.

Appendix A: Glossary 136
12 Revision History 137
Notices: 138

Table of Figures

Figure 1 Documentation Code Sample 10
Figure 2 LLRP Messages 17
Figure 3 LLRP Basic Field Types 18
Figure 4 Sample LLRP Application Flow 23
Figure 5 UTCTimestamp Parameter Copyright 2006, 2007 EPCglobal, Inc. 25
Figure 6 LTK Parameter Machine Description 26
Figure 7 LTK Vencor Extension Machine Description 27
Figure 8 LLRP Binary SET_READER_CONFIG Message 27
Figure 9 Annotated SET_READER_CONFIG Message 27
Figure 10 LTK-XML SET_READER_CONFIG Message 28
Figure 11 LTKC – Sample Compilation and Execution 31
Figure 12 LTKCPP – Linux Makefile Sample Compilation and Execution 32
Figure 13 LTKCPP – Visual Studio Sample Compilation and Execution 33
Figure 14 LTKNET – Sample Compilation 35
Figure 15 LTKNET – Sample Execution 36
Figure 16 LTKJAVA – Windows Library Extraction 36
Figure 17 LTKJAVA – Windows Sample Compilation 38
Figure 18 LTKJAVA – Windows Sample Execution 39
Figure 19 LTKJAVA – Linux Library Extraction 39
Figure 20 LTKJava – Linux Sample Compilation 40
Figure 21 LTKJava – Linux Sample Execution 41
Figure 22 LTKCPP CConnection Class 42
Figure 23 LTKCPP CElement Class 42
Figure 24 LTKCPP/LTKC Element Tree 43
Figure 25 LTKCPP Accessor Methods 44
Figure 26 LTKCPP Sequence to Determine Element Type 45
Figure 27 LTKCPP Constructing and LTK-XML Decoder 46
Figure 28 LTKCPP Decoding using an LTK-XML Decoder 46
Figure 29 LTKCPP Destroying an LTK-XML Decoder 46
Figure 30 LTKCPP Converting a Message to a Known Type 47
Figure 31 LTKCPP CErrorDetails Class 47
Figure 32 LTKCPP– Error Reporting Example 48
Figure 33 LTKNET – Core Application Classes 50
Figure 34 LTKNET – Core Message Classes 51
Figure 35 LTKNET – Core RoSpec Classes 52

Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 36 LTKNET – Core AccessSpec Classes 53
Figure 37 LTKJAVA – Implementing LLRPEndpoint MessageReceived 54
Figure 38 LTKJAVA – Application Initiated Reader Connection 55
Figure 39 LTKJAVA – Constructing an LLRP Message from LTKJAVA Objects 55
Figure 40 LTKJAVA – Constructing an LLRP Message from LTK-XML 56
Figure 41 LTKJAVA – Iterating Custom Sub-Parameters 56
Figure 42 LTKJAVA – Processing Custom Sub-Parameter 57
Figure 43 LTKC – Compiling and Linking 58
Figure 44 LTKC – Compiling and Linking with Libxml2 58
Figure 45 LTKCPP – Compiling and Linking 58
Figure 46 LTKCPP – Compiling and Linking with Libxml2 59
Figure 47 LTKNET – Compiling and Linking 60
Figure 48 LTKJava – Import Statement Example 60
Figure 49 LTKCPP – Initailizing 61
Figure 50 LTKNET – Initializing 61
Figure 51 LTKJAVA – Initializing 61
Figure 52 LTKCPP – Creating Connection 62
Figure 53 LTKCPP – Validating Connection 64
Figure 54 LTKNET – Creating/Validating Connection 64
Figure 55 LTKJava – Creating/Validating Connection 65
Figure 56 LTKCPP – Enabling Impinj Extensions 67
Figure 57 LTKNET – Enabling Impinj Extensions 67
Figure 58 LTKJAVA – Enabling Impinj Extensions 68
Figure 59 LTKC – Printing LLRP Messages 69
Figure 60 LTKCPP – Printing LLRP Messages 69
Figure 61 LTKNET – Printing LLRP Messages 70
Figure 62 LTKJava – Printing LLRP Messages 70
Figure 63 LTK-XML – Basic Application Sequence 72
Figure 64 LTKCPP – Factory Default LLRP Configuration 73
Figure 65 LTKNET – Factory Default LLRP Configuration 74
Figure 66 LTKJava – Factory Default LLRP Configuration 75
Figure 67 LTK-XML – Simple ROSpec 76
Figure 68 LTKCPP – Constructing a Simple ROSpec 79
Figure 69 LTKNET – Constructing a Simple ROSpec with Objects 81
Figure 70 LTKJava – Constructing a Simple ROSpec with Objects 82
Figure 71 LTKCPP – Enable an ROSpec 84
Figure 72 LTKNET – Enable an ROSpec 84
Figure 73 LTKJAVA – Enable an ROSpec 85
Figure 74 LTKCPP – Start an ROSpec 86
Figure 75 LTKNET – Start an ROSpec 87
Figure 76 LTKJava – Start an ROSpec 88

Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 77 LTKCPP –Receiving Asyncrhonous Reader Messages 89
Figure 78 LTKCPP – Extracting Individual Tag Reports 89
Figure 79 LTKNET –Adding Event Handlers for Asyncrhonous Reader Messages 90
Figure 80 LTKNET –Receiving Asyncrhonous Reader Messages 91
Figure 81 LTKJava –Receiving Asyncrhonous Reader Messages 92
Figure 82 LTKJava –Extracting Data From Tag Reports 93
Figure 83 LTKNET – Retreiving Device Capabilities 96
Figure 84 LTKCPP – Retreiving Device Capabilities 99
Figure 85 LTKJAVA – Retreiving Device Capabilities 100
Figure 86 LTKXML – Impinj Octane LLRP Extensions 105
Figure 87 LTKNET – Setting Configuration with LTK-XML 107
Figure 88 LTKCPP – Importing SET_READER_CONFIG from LTK-XML File 108
Figure 89 LTKCPP – Setting Antenna Configuration and Transmit Power 109
Figure 90 LTKJAVA – Setting Antenna Configuration and Transmit Power 110
Figure 96 LTKXML – SET_READER_CONFIG for docSample3 113
Figure 97 LTKCPP – SET_READER_CONFIG for docSample3 114
Figure 98 LTKNET – InventoryFilters for docSample3 115
Figure 99 LTKJAVA – InventoryFilters for docSample3 117
Figure 100 LTKXML –AccessSpec to read Tag User Memory 119
Figure 101 LTKNET –Adding an AccessSpec with LTK-XML 120
Figure 102 LTKNET –Adding an AccessSpec with C# Objects 122
Figure 103 LTKNET –Processing AccessSpec results in TagReportData 123
Figure 104 LTKCPP –Creating an AccessSpec 124
Figure 105 LTKCPP –Processing AccessSpec results in TagReportData 125
Figure 106 LTKJAVA –Creating an AccessSpec 126
Figure 107 LTKJAVA –Processing AccessSpec results in TagReportData 126
Figure 108 LTKNET –Trigger Asyncrhonous reports with GET_REPORT 127
Figure 109 LTKCPP –Trigger Asyncrhonous reports with GET_REPORT 127
Figure 110 LTKJAVA –Trigger Asyncrhonous reports with GET_REPORT 127
Figure 111 LTKXML –Setting Tag Report Contents 130
Figure 112 LTKCPP – Setting Tag Report Contents 131
Figure 113 LTKCPP – Extracting Impinj Tag Report Data 132
Figure 114 LTKNET – Extracting Impinj Tag Report Data 132
Figure 115 LTKCPP – Adding Custom OpSpec 134
Figure 116 LTKNET – Adding Custom OpSpec 134
Figure 117 LTKCPP – Retreiving Custom OpSpecResults 134
Figure 118 LTKNET – Retreiving Custom OpSpecResults 135

Copyright © 2009, 2010, 2011 Impinj, Inc.

1 Introduction

1.1 Purpose of Guide

This guide provides the experienced programmer with the information necessary to write RFID applications communicating with Speedway Revolution™ readers via Octane™ LLRP.

1.2 Intended Audience

This guide provides software and/or systems engineers with information for creating applications that will communicate with and control Impinj Readers via LLRP. Prior programming knowledge in C++ 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, document formatting, and both necessary and optional references.
- Section 2 provides information about the Low-Level Reader Protocol (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 a Speedway Reader.
- Section 7 provides the process for creating a simple Impinj LTK application
- Section 8 details the usage of the 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.
- Appendix A: Glossary provides a glossary of common terms used in this guide.

1.4 Document Conventions and Terms

Table 1Document Style Conventions

Type	Example	Style
LLRP message	INJ_ENABLE_EXTENSIONS	CAPS _UNDERSCORES
LLRP parameter	*AntennaConfiguration*	***ItalicsCamelCase***
LLRP field	ResetToFactoryDefault	**UnderlineCamelCase**
Enumerated field value	'Upon N Tags or End of AISpec'	'Single-Quoted String'
File name	**'ImpinjDef.xml'**	'single-quoted courier bold'
LTK function	***getLLRPStatus***	Courier bold italics case matches programming syntax

Copyright © 2009, 2010, 2011 Impinj, Inc.

Type	Example	Style
LTK class names	**CIMPINJ_TCS_RESPONSE**	Courier bold: case matches programming syntax

Additional style usage:

- Important LLRP and LTK usage notes are shown in *italics* in a separate paragraph which is begins with **Impinj Best Practice!**
- Code examples are formatted in separate text boxes using Courier font. . 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 the 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 solving these problems. These sections are shown in code comments via the following bold 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 Documentation Code Sample

Copyright © 2009, 2010, 2011 Impinj, Inc.

1.4.1 Document Terms

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

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

Custom Extension: An LLRP mechanism that allows vendors to add functionality beyond standard LLRP behavior.

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

EPCglobal: EPCglobal leads the development of industry driven standards for the Electronic Product Code (EPC) to 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, extended, compiled, tested, and packaged for use with Speedway Revolution Readers by Impinj.

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

STL: Standard Template Library.

1.5 Technical Support

For technical support issues related to LLRP, the Impinj LTK, or using of Octane or Speedway, 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 software downloads, Impinj LTK releases, additional code samples, and FAQs describing the use of Speedway Revolution Readers 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 does not provide sufficient detail, the best sources for additional information are this 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

this document into the Impinj LTK. For your convenience, the links in the following sections access the full LLRP standards documents.

- Section 6 provides the LLRP standard describes theory of operation in 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 as 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 via 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 via 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 and the unique Octane LLRP custom extensions that provide additional capabilities. This guide provides detailed information to developers who are planning to support Impinj 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 Revolution 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 using the LTK, and open source tools and documentation to accompany LTK or LLRP. This site does not contain any information on 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, including Perl and Java.

It is possible to build the all open-source llrp-toolkit with support for Impinj extensions. Those requiring a language not formally released by Impinj, desiring special modifications to the LTK, or those wishing to have source visibility for the toolkit are encouraged to do so. The LTK wiki hosted at http://llrp-toolkit.wiki.sourceforge.net is significant and beneficial. It contains useful information on the LTK architecture and operation.

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 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 Speedway Revolution Reader 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 Speedway Revolution Reader, please refer to the *Octane LLRP Reference Guide* and Application Note, entitled *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. Having already

Copyright © 2009, 2010, 2011 Impinj, Inc.

standardized the tag and reader radio frequency (RF) air interface protocol with UHF Gen 2, the LLRP specification was the practical, logical next step in facilitating 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 the UHF Gen 2 air protocol standard definition, Impinj assumed a leadership role, dedicating important internal resources who worked closely with other companies to both define the standard and work toward open source programming support.

The device operating an RFID Reader can vary from a software application to middleware running on server hardware to programmable logic controllers. While the ultimate end-business application these clients support may differ, 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 may 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 critical to technology innovation and wide-spread technology adoption. LLRP, by removing the task of writing numerous proprietary protocol interfaces that all support the same functionality, LLRP has let controlling application software vendors 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 using responses, and also generates asynchronous data called events and reports.

LLRP supports Reader or client initiated connections. By default Speedway Revolution 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. On Speedway Revolution, 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.

The Speedway Revolution Readers only allow one LLRP connection at any time. Attempts to connect when an existing connection is active and healthy will result in a standard LLRP message back to the application reporting that another connection is currently active.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 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 store inside the message as either fields or parameters. Figure 2 contains a list of all LLRP message (except for responses), whether the message will solicit a response, which side of the LLRP connection initiates the message, and a short description of the message purpose.

2.4.1 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

Message Name	Resp	Client/ Reader	Description
			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 and *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
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)

Copyright © 2009, 2010, 2011 Impinj, Inc.

Message Name	Resp	Client/ Reader	Description
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

Figure 2 LLRP Messages

2.4.2 Fields

Fields are individual data elements with a known format. LLRP has ten basic types listed in Figure 3 below for reference. In addition to these basic types, LLRP defines fields that contain lists (or vectors) of some these basic types, these are also noted in Figure 3 below. When constructing LLRP messages/parameters, all fields must be present and within their valid range, although some fields are conditionally ignored.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 mostly for 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

Figure 3 LLRP Basic Field Types

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. Speedway Revolution supports one *ROSpec* on the reader at a time. A *ROSpec* contains the following elements. Unless otherwise noted, all elements are mandatory.

Copyright © 2009, 2010, 2011 Impinj, Inc.

- *ROSpecID* – an ID set by the client application to uniquely identify this spec in subsequent commands.
- Priority – Currently must be zero
- 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 Readers 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* triggers 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 Reader 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 antennas ports (that have antennas connected) and uses the default Gen2 and RF configuration as specified in the SET_READER_CONFIG message. However, a *ROSpec* may contain multiple *AISpecs*. Program multiple *AISpecs* into the Reader for more complex behavior to be executed autonomously and without

Copyright © 2009, 2010, 2011 Impinj, Inc.

intervention from the application. An application may 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 Revolution Readers 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 may be 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. An option for programming is via two *AISpecs* within a single *ROSpec* which executes automatically by the Reader.

The application requires different Gen2 parameters at different times. One example may be 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 search for all tags (or all case tags). Programming this scenario via two *AISpecs* within a single *ROSpec* and automatically processed by the Reader is an option.

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

- The Reader applies the configuration contained in the first *AISpec* as soon as the *ROSpecStartTrigger* 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 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*.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 generate and are sent to the client application.

AccessSpecs design supports multiple, different air protocols with unique commands. Because the Speedway 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 which 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 Revolution 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 which 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* to apply during inventory. is enabled,.
- 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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*.

Some examples 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 may be helpful for debugging and testing the application.*

2.6 A typical LLRP Exchange

A typical LLRP application involves these steps:

1. Application or 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 on 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 configuration of 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, it is recommended to restore the LLRP factory defaults when connecting to a Reader.*

5. Application adds and enables a ROSpec.
6. Application adds and enables multiple AccessSpecs (optional).
7. Application triggers *ROSpec* start and stop (optional).
8. Reader sends requested data report to application.

A LTK example of this procedure is described in Section 6.5.3. A sample timeline of this exchange (steps 5-8) displays in Figure 4.

Copyright © 2009, 2010, 2011 Impinj, Inc.

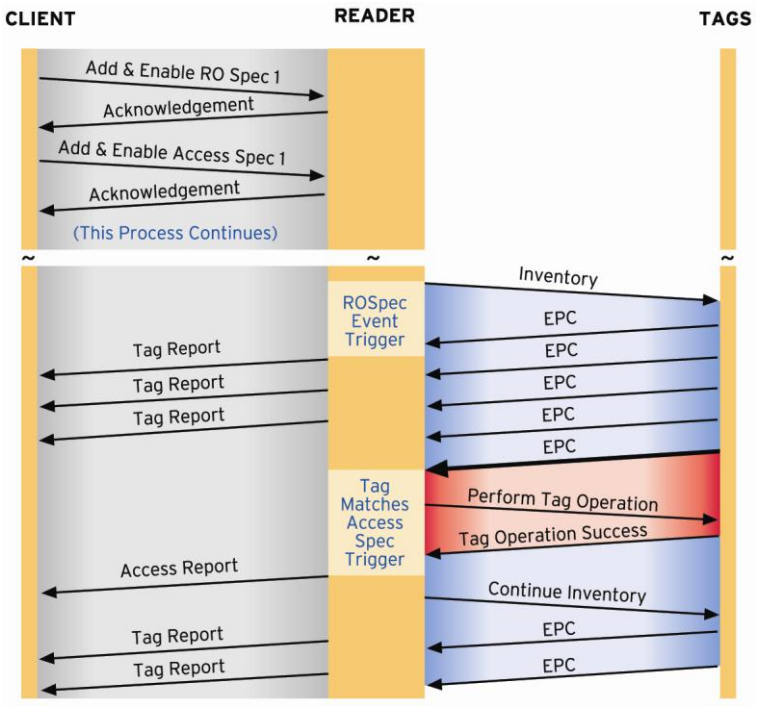


Figure 4 Sample LLRP Application Flow

2.7 Vendor Extensions to LLRP

Other standards have been proposed for the Reader network interface, 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. The 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 that allows 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 Impinj 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 Speedway 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 Speedway Revolution without any Impinj Octane extensions to LLRP, learning to use the extensions available via Octane LLRP enables advanced feature to make configuration easier and extracts more performance from your RFID application.

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

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, requiring 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 for 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, performing message transactions that automatically wait for the Reader's response, and 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

using the pre-built Impinj LTK, these definition files have been integrated into the release packages. Sub-section 3.3 and 3.4 are not necessary reading or 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 and LTKNET process this definition file at compile time to build the classes and methods to support building and parsing 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]																															

Figure 5 UTCTimestamp Parameter

Copyright 2006, 2007 EPCglobal, Inc.

These two illustrations 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 here is in 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 an 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 6 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 are 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
  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 7 LTK Vencor 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 tech 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 three 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 8 LLRP Binary SET_READER_CONFIG Message

This is difficult to read however and troublesome to interpret or change. To make it human readable one could add some annotations such as

```
LLRP-HEXBINARY (annotated)
04 | 03 | 00 00 00 13 | 00 00 00 00 | 00 | 00 E1 | 00 08 | 00 03 | 80 | 02
Hdr  Msg       Length      MessageID  Fac   type     Len     GPI#  Cfg  State
```

Figure 9 Annotated SET_READER_CONFIG Message

which 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:

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
LTK-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 10 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 involving LLRP message and parameters will show the LTK-XML version of the parameters since they are readily human readable and usable by some of the LTK libraries directly. 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 Revolution occurs in binary form.

Impinj Best Practice: *When submitting support requests while programming with the LTK, we recommended including the LTK-XML copies of your ROSpecs and AccessSpecs. This provides needed information for our support team to analyze your issue.*

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 requiring LLRP support. Impinj distributes periodic releases of this toolkit. They are named **`'libltkc_sdk_X_Y_Z.tgz'`**. The release includes static Linux libraries for x86 and Arm platforms. The Arm library is compatible with Speedway Revolution Readers. The x86 library is compatible with most 2.6 Linux systems and is useful 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 requiring 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 Revolution Readers. 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 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 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 (over **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 this file before starting integration with an existing application.*

Copyright © 2009, 2010, 2011 Impinj, Inc.

4.1.3 Versioning

In the example release names above, "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 with previous version backward compatibility. . "Z" denotes the maintenance release version. Changes in the maintenance version represents fully backward compatible bug fixes and do not affect the programming APIs. Impinj recommends for all new updates of the LTK, a re-compile and re-link of your application.

4.2 Obtaining the LTK

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

4.3 LTK Required Tools/Environment

Compiling your application and linking to the SDK requires compiler tools.

LTKC –x86 Linux requires gcc. On-reader development requires xScale embedded SDK based on gcc or 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 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 your Reader connects to the network and accessible by hostname from your target application machine.

For details on 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 within in a subdirectory libltkxxx (libltkc, libltkcpp, libltkjava, and libltknet) where xxx is different for each library. Some libraries contain large directories of html based documentation within the libltkxxx folder. Program compilation does not require these libraries.

Each library contains sample code to build and test the libraries on your system. Impinj strongly recommends that the libraries functions validate by running compile and the sample code on your system. Instructions below display building and running 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. 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
[pdietric@woodbridge 10_6_0]$ cd libltkc
[pdietric@woodbridge libltkc]$ cd example
[pdietric@woodbridge example]$ make
cc -g -I.. -o example1 example1.c ../libltkc_x86.a
[pdietric@woodbridge example]$ ls
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
example1  example1.c  Makefile
[pdietric@woodbridge example]$ ./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
[pdietric@woodbridge example]$
```

Figure 11 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 Speedway Revolution 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 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 Speedway Revolution embedded arm platform, and everything to compile for all targets. Executable applications build in the docSampleX subdirectories and use suffixes describing their target platform.

```
LTKCPP Linux
[pdietric@woodbridge libltkcpp]$ make all_x86
cd docsample1;  make all_x86
make[1]: Entering directory `/home/pdietric/Development/Temp/libltkcpp/docsample1'
make all AR=ar CXX=g++ CPPFLAGS="-g -Wall -I.." SUFFIX=_x86
```

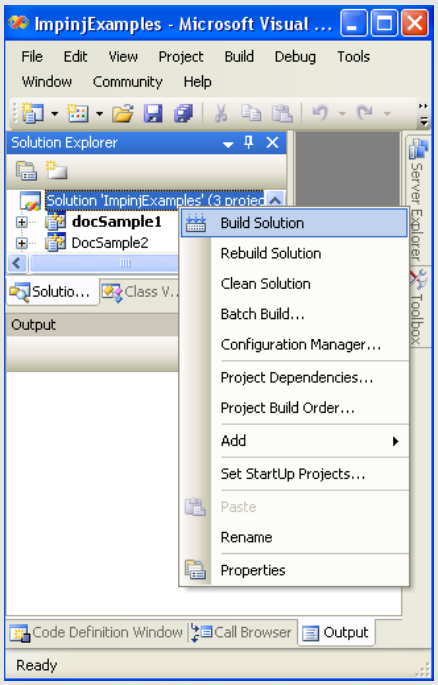

Copyright © 2009, 2010, 2011 Impinj, Inc.

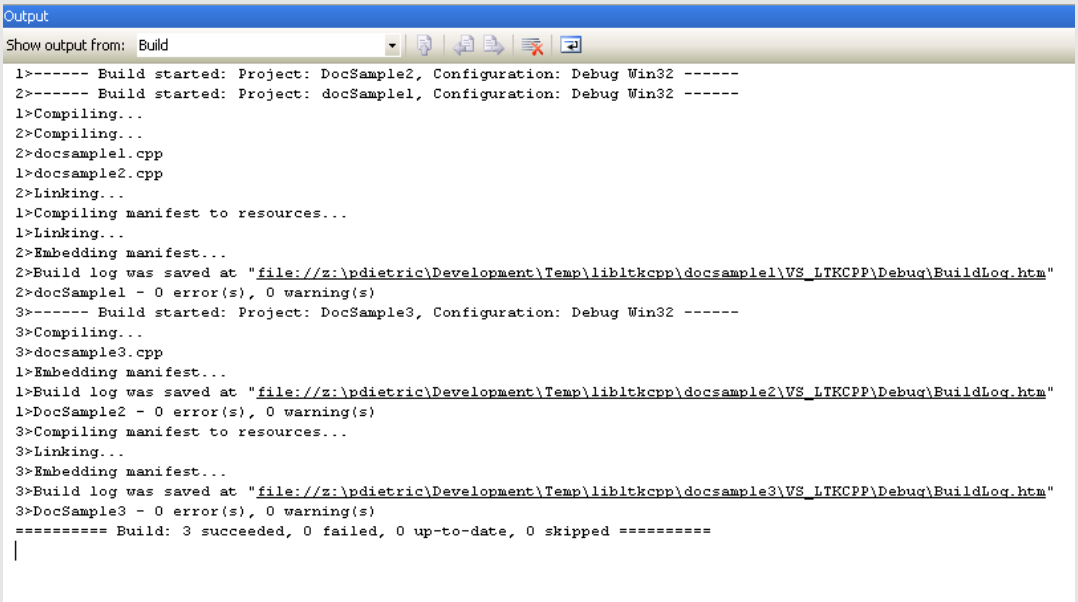
```
…
make[1]: Leaving directory `/home/pdietric/Development/Temp/libltkcpp/docsample3'
[pdietric@woodbridge libltkcpp]$
```

Figure 12 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





Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 13 LTKCPP – Visual Studio Sample Compilation and Execution

Application can be run from the command line in either DOS or Linux.

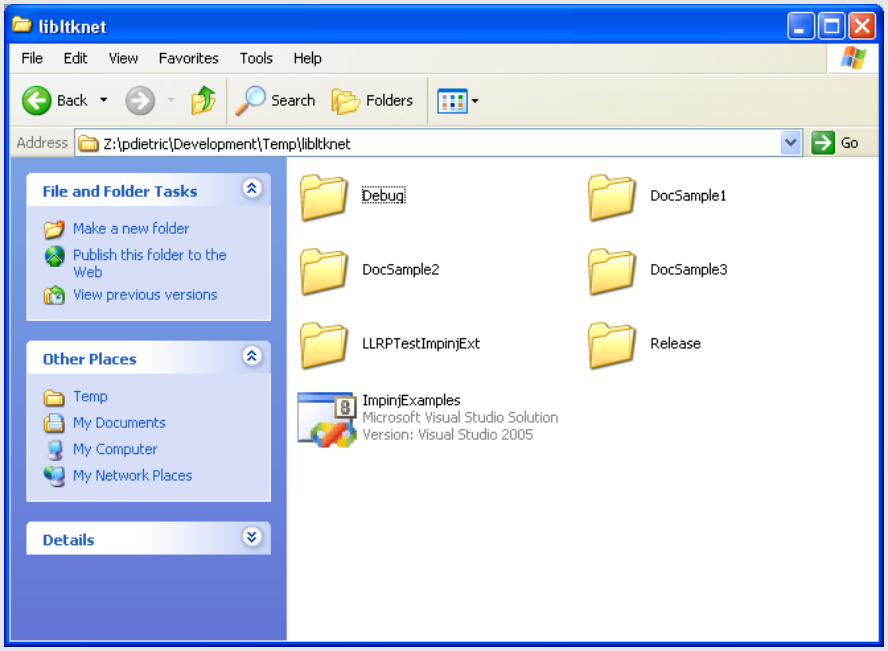
```
LTKCPP Docsample1-x86

[pdietric@woodbridge libltkcpp]$ cd docsample1/
[pdietric@woodbridge 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

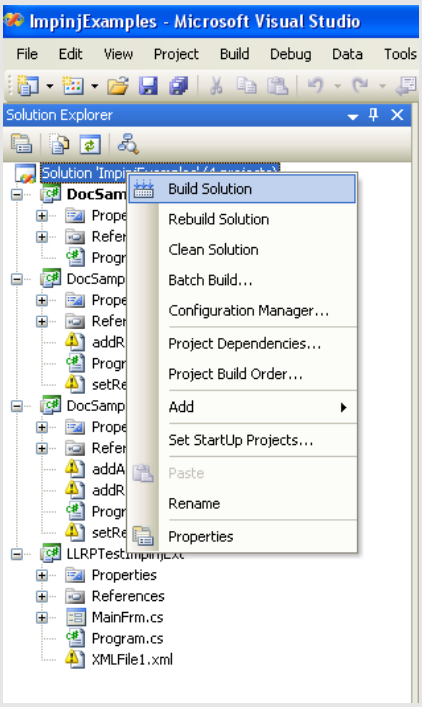
Unzip the archive file and browse to the **'libltknet'** directory. Open the **'ImpinjExamples.sln'** solution file. An LTKNET example displays on the next page.

Copyright © 2009, 2010, 2011 Impinj, Inc.

LTKNET



Build the samples by right-clicking the solution or by using the Build tab at the top of the IDE.



Copyright © 2009, 2010, 2011 Impinj, Inc.

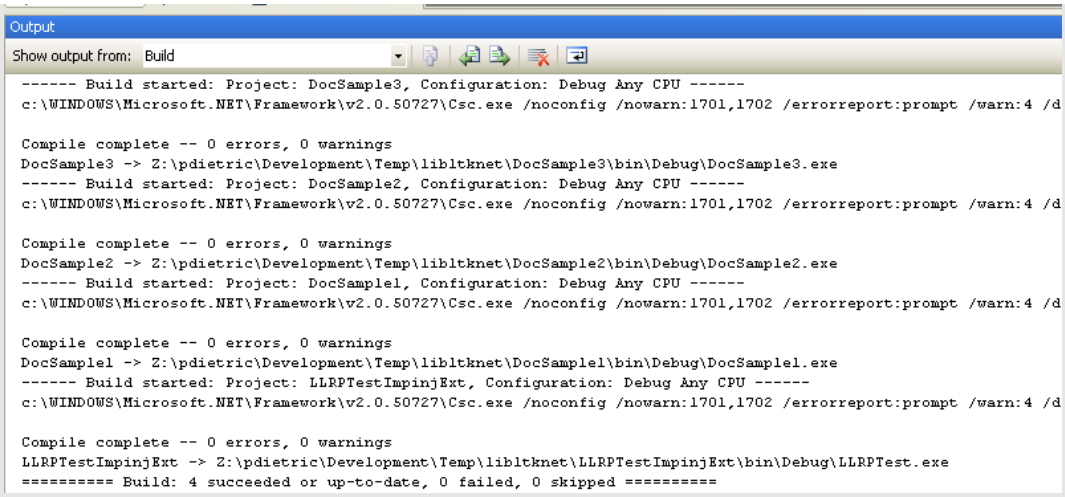


Figure 14 LTKNET – Sample Compilation

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

```
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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
AAAAAAAA
EEEEEEEE
 . . .
```

Figure 15 LTKNET – Sample Execution

4.4.4 LTKJava

4.4.4.1 Windows

Download the zip file from http://support.impinj.com. Extract the files from the archive.

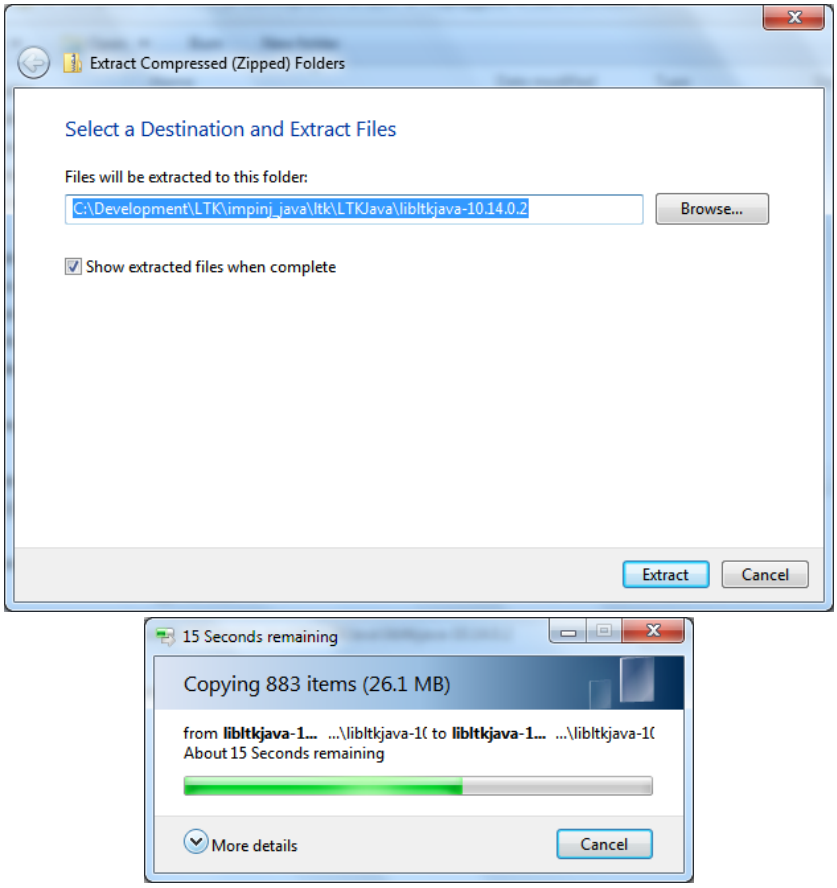
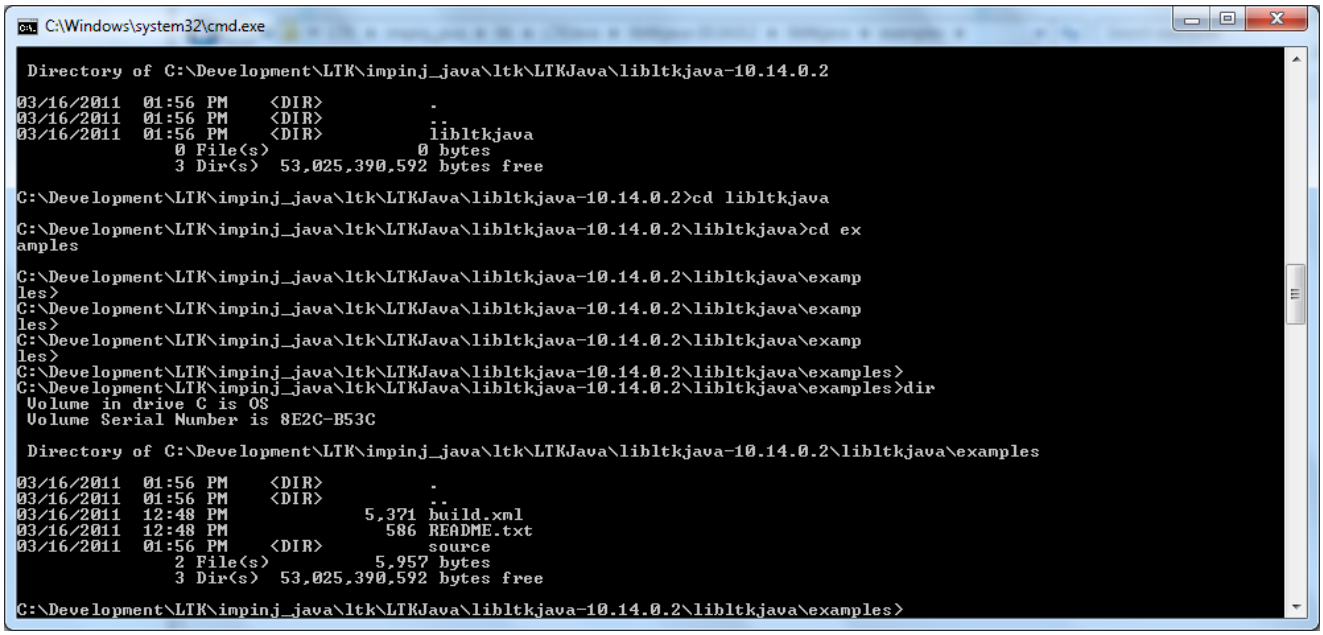


Figure 16 LTKJAVA – Windows Library Extraction

Copyright © 2009, 2010, 2011 Impinj, Inc.

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



Setup your path to ensure access to both JAVA and ant. Your path may look something like this. You may also need to set your JAVA_HOME environment variables. See examples below:

```
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:
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
     [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 17 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\exampl
es\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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

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

Figure 18 LTKJAVA – Windows Sample Execution

4.4.4.2 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 19 LTKJAVA – Linux Library Extraction

NOTE: The version number of the file you download reflects the current version of LTKJAVA.
Change into the **`libltkjava`** directory. Review the **`CHANGES.TXT`** file to view the latest known issues and revisions to LTKJAVA. Change into the examples directory.

```
LTKJAVA

cd libltkjava

cd examples
```

Set up your ant path and your ANT_HOME and JAVA_HOME environment variables.
NOTE: Your values will likely be different than these, 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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 20 LTKJava – Linux Sample Compilation

Running an example requires a Speedway Revolution 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. 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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
     [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 21 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, 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 message, receive messages, perform transactions (a send with an expected reply) and get error status. Examples using the **`CConnection`** class are shown in Section 6.

Copyright © 2009, 2010, 2011 Impinj, Inc.

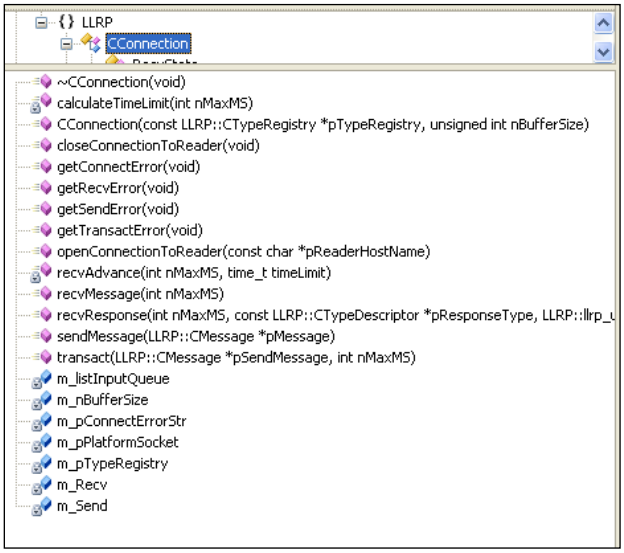


Figure 22 LTKCPP CConnection Class

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

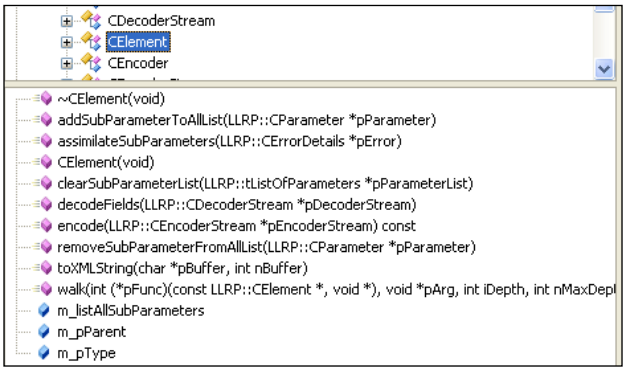


Figure 23 LTKCPP CElement Class

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, may in turn reference zero or more subparameters, and so on. An element that has subparamters is said to be the enclosing **`CElement`** of each sub-parameter. For example, in Figure 24, element X is the enclosing element of A, B1, B2, B3, and C.

[1] 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

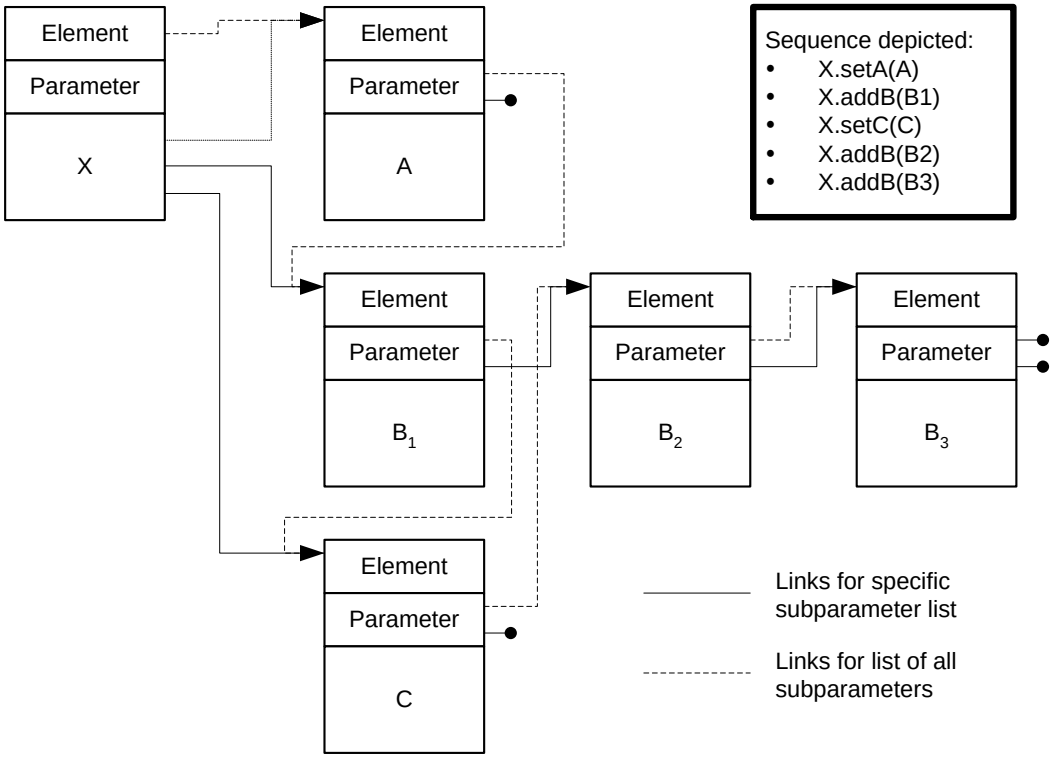


Figure 24 LTKCPP/LTKC Element Tree

Elements maintain an all-list, which is a list of all sub-parameters enclosed by the element. The all-list is important to 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 24.

5.1.3 Fields, Subparameters, and Accessor Functions

A **`CElement`** may contain:

- Simple fields, such as integer or enumerated values. For these fields there are accessor functions ***`getfield()`*** and ***`setfield()`***. Vector fields, such as a string or a vector of 32-bit frequencies. For these there are 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 from the all-list and destructs it, if there is one. It then adds the new subparameter to the all-list and sets the specific pointer. The ***`setsubparam()`*** function returns a status code, OK (0) for success, or 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 and the application must determine the actual type and cast accordingly. (See section 5.1.5 How to Determine the Type of an Element.)

Lists of subparameters, use for mandatory (1-N) and optional (0-N) lists. For these subparameters there are accessors ***`beginsubparameter()`***, ***`endsubparameter()`*** for C++,

Copyright © 2009, 2010, 2011 Impinj, Inc.

nextsubparameter() for C, ***and addsubparameter().*** The ***addsubparameter()*** accessor links the subparameter to the all-list. The ***addsubparam()*** function returns a status code, OK (0) for success, or a specific code for an error. In C++ uses the Standard Template Library (STL) lists . In C there is no such intermediate data structure. In both cases, like 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 includes 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 25 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 should be considered mandatory. In C, it is possible to use local instances (as does the example) by using great care

The reason is 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 causes havoc as described below.

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 proper memory management in your application.

Copyright © 2009, 2010, 2011 Impinj, Inc.

Impinj Best Practice! *Impinj strongly recommends 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 determines at run-time. For example, a notification message arrives as an RO_ACCESS_REPORT, a READER_EVENT_NOTIFICATION, or a KEEPALIVE. Another example within the *ROSpecs* lists *SpecParameters*, each may 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. Determine the element type by comparing the type label pointer to the address of a known type descriptor. Once the type is known, use casting for the specific element type.

In the examples see 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 26 LTKCPP Sequence to Determine Element Type

5.1.6 Practical Information on Building Message in LTKCPP

Build LTKCPP messages using these working tips.

LTKCPP uses standard naming for all of its class names, methods, and members. After constructing the message with the ***new*** operator, the programmer looks at all of the members of the name ***addXXX*** or ***setXXX***. Set all of the public LLRP parameters and fields via the LTK. Set functions that take a basic type require no allocation of memory. Sub-parameter set functions which take a pointer as an argument require the sub-parameter to be created via another new operation. Add functions signify that LLRP allows zero or more if 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 are sometimes optional in LLRP. The LTK will not provide any warnings or errors when mandatory sub-parameters are missing. The Reader returns these errors. Building Messages from LTK-XML

Copyright © 2009, 2010, 2011 Impinj, Inc.

Build an LTKCPP message object using LTK-XML, following these steps:

1. Create and validate an LTK-XML decoder using the type Registry 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 27 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 28 LTKCPP Decoding using an LTK-XML Decoder

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

```
LTKCPP

    delete pDecoder;
```

Figure 29 LTKCPP Destroying an LTK-XML Decoder

4. To modify the message, convert 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 */
    }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

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

Figure 30 LTKCPP Converting a Message to a Known Type

5.1.7 Error Reporting (ErrorDetails Class)

Error reporting via CErrorDetails structure whereby LTKC/LTKCPP provides error information to the application:

- A status code, 0 always represents OK. The status code is enumerated and the application easily distinguishes errors. Error codes define by an enumeration in 'ltkc_base.h' or 'ltkcpp_base.h'.
- A string error message, 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 may access them directly.

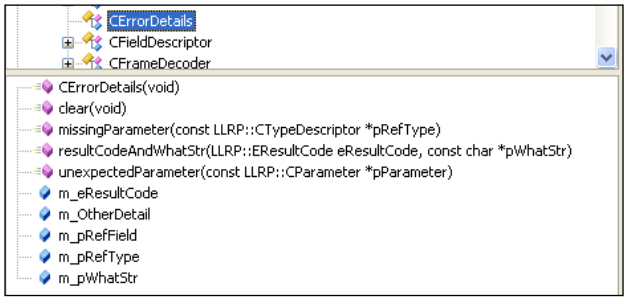


Figure 31 LTKCPP CErrorDetails Class

In the example program see the ***transact()*** function. A portion that detects an error 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)
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        {
            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 32 LTKCPP– Error Reporting Example

5.1.8 Extensions

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

5.1.8.1 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. Consider GET_READER_CAPABILITIES_RESPONSE: It may contain custom parameters within reader capabilities which 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 the LLRP use cases. Assume the application sends a custom parameter in a SET_READER_CONFIGURATION that the Reader does not understand. It is a safe conclusion the Reader functions will not act as the application intends. Normally the reader responds with an error.

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

5.1.8.2 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.

To summarize, constructing and manipulating a custom message or parameter is similar 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.

5.1.8.3 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

creates a list of CParameters (similar to choices) and names the subparameter list Custom. The custom list accessor functions use pointers to the generic CParameters type (see section 5.1.3 Fields, Subparameters, and Accessor Functions).

If the application tries to add a *CustomParameter* to an extension point (also known as Custom) list that is not allowed at that point, the ***addCustom()*** accessor function returns an error. See Figure 25 for an example of the custom parameter accessors.

When the application scans the extension point (aka *CustomParameter*) list, it must check each subparameter type (see section 5.1.5 How to Determine the Type of an Element) before (optionally) processing the subparameters.

5.1.8.4 Enrollment in TypeRegistry

Each LTK definition file processes the LTKCPP enrolment function when compiling the libraries. The enrollment function takes as an argument, a ***typeRegistry*** 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 the 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.

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 LTKNET via a single. This class contains methods to open and close the Reader connection; transaction method which sends LLRP messages and awaits the responses. Event handlers handle asynchronous incoming messages from the Reader.

Copyright © 2009, 2010, 2011 Impinj, Inc.

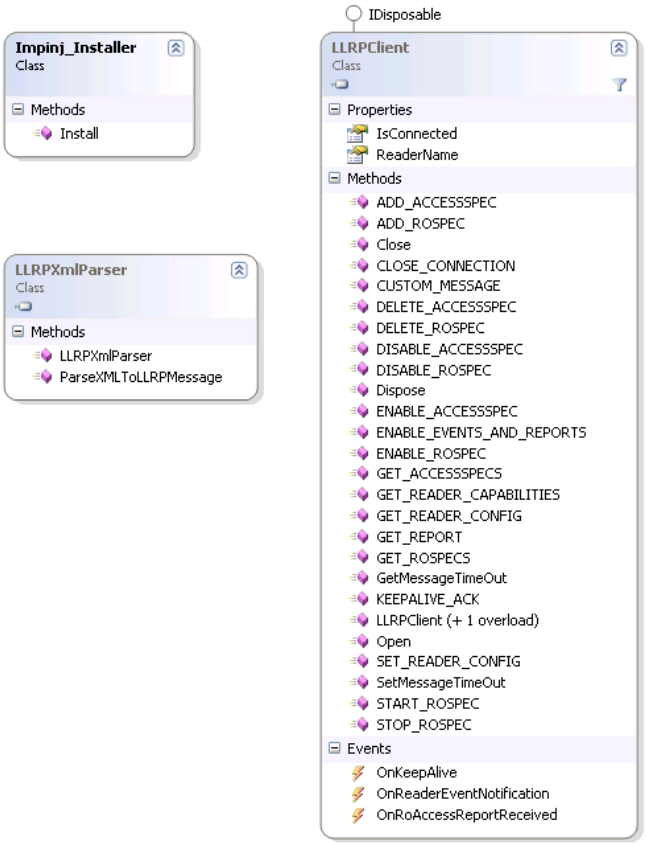


Figure 33 LTKNET – Core Application Classes

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 message directly into C# objects. See Figure 33 for details.

5.2.3 LLRP Message Classes

Figure 34 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. Performing basic LLRP operations requires the following primary message classes.

Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 34 LTKNET – Core Message Classes

5.2.4 Handling ROSpecs

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

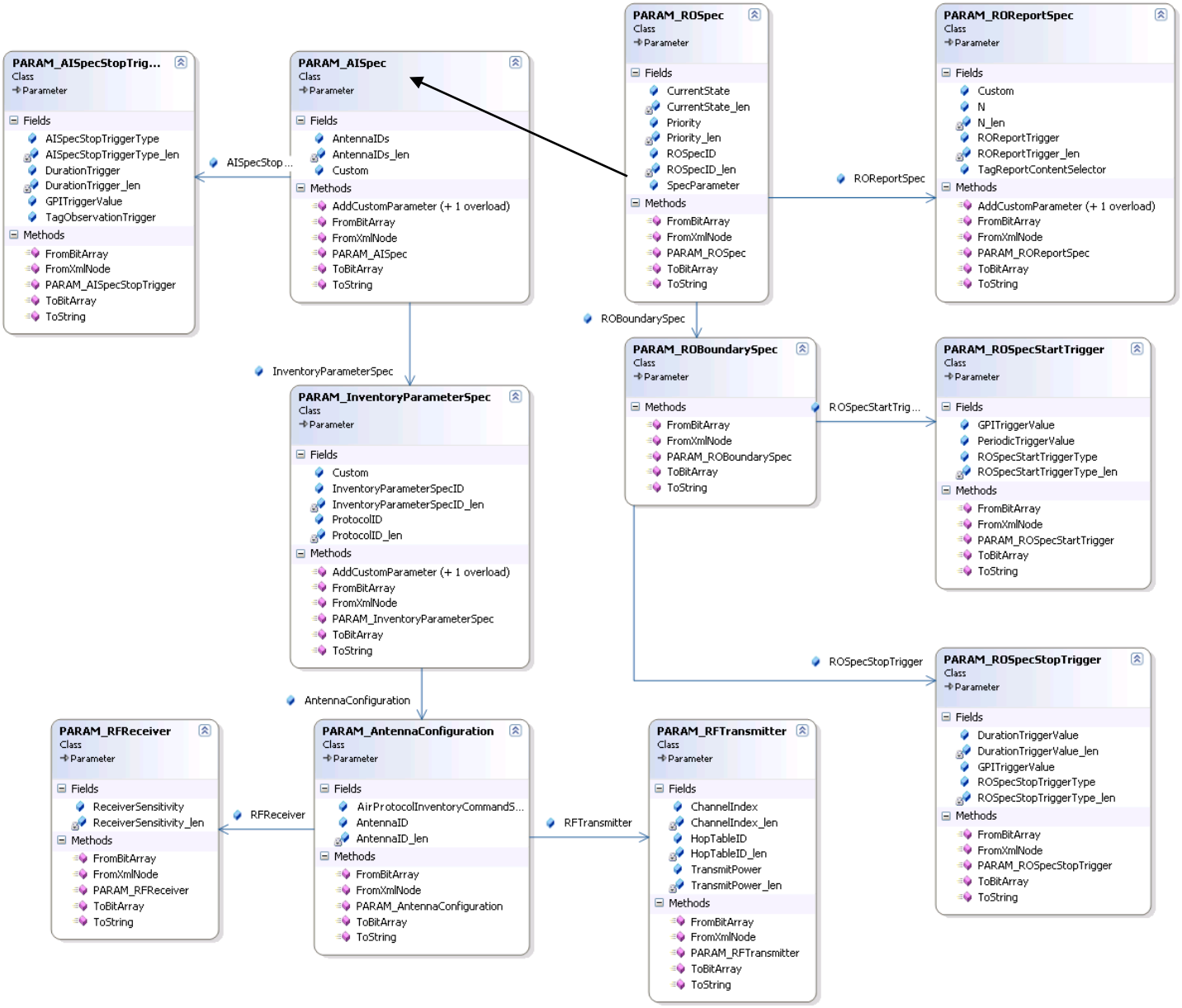


Figure 35 LTKNET – Core RoSpec Classes

5.2.5 Handling AccessSpecs

Key parameters for building a *PARAM_AccessSpec* display below. 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*.

Copyright © 2009, 2010, 2011 Impinj, Inc.

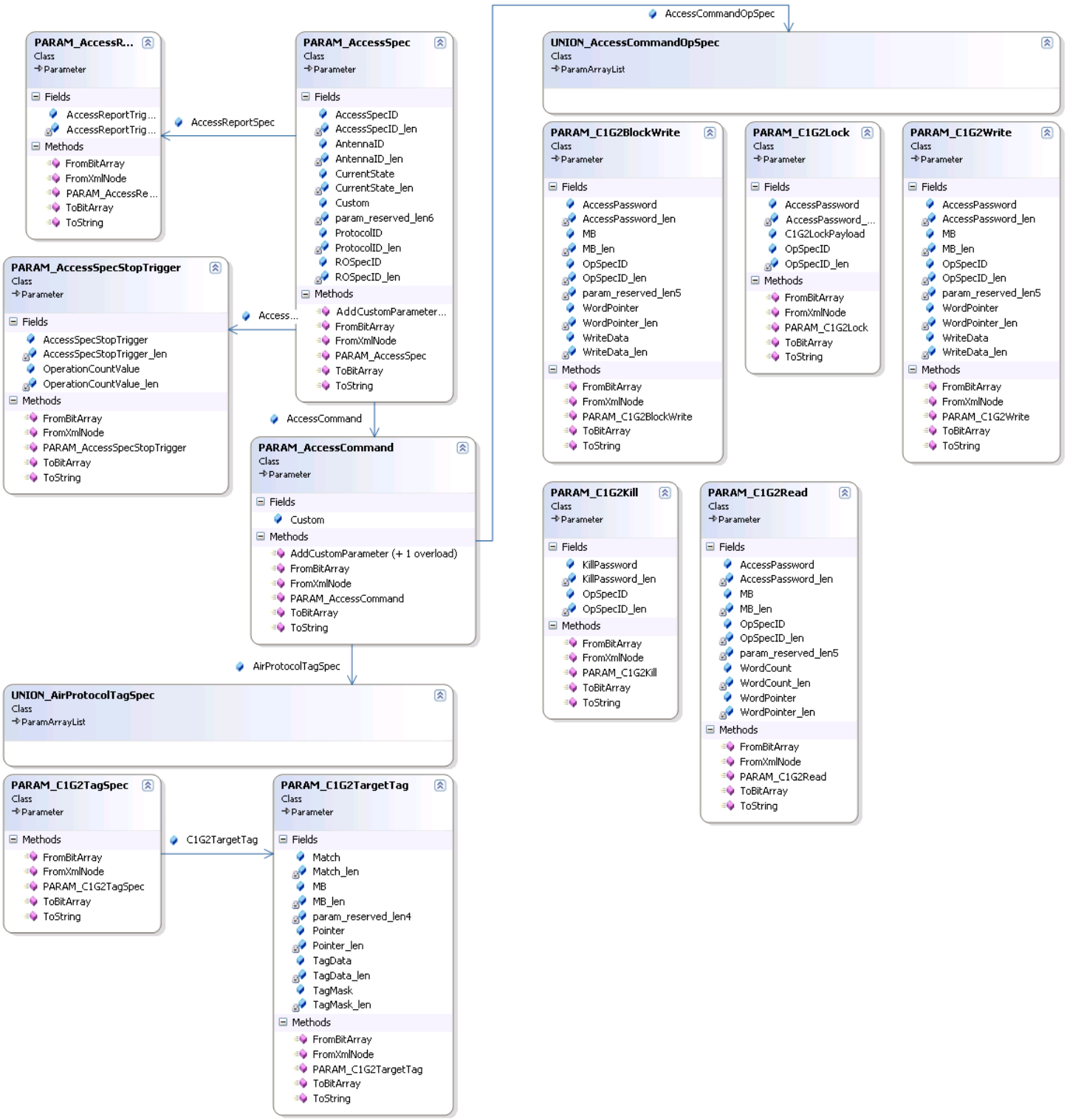


Figure 36 LTKNET – Core AccessSpec Classes

5.3 LTKJAVA

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 message 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 following example 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 asychronously");

    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 37 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 Reader initiates. 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 (also derived from **LLRPConnection**).

Below is an example of creating a connection to a reader with the **LLRPConnector** class. This shows that connection failures are returned to the application through the ***LLRPConnectionAttemptFailedException*** event.

```
LTKJAVA

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


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
// 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 38 LTKJAVA – Application Initiated Reader Connection

5.3.3 Sending and Receiving Messages

Once connected, the application sends and receives messages through the **LLRPConnection** object (parent) returned by the **LLRPConnector** constructor. When sending messages, the **LLRPConnection** API provides two different methods: 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 send and receive **LLRPMessage** objects. Discussion about **LLRPMessages** and their definitions follow.

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 to build **LLRPMessage** objects. First, they can be constructed via create the individual parameters and adding them to the message via Java methods. Second, XML strings or files can create LLRP messages.

Constructing simple messages via LTKJAVA objects is typically easiest. Below is 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 39 LTKJAVA – Constructing an LLRP Message from LTKJAVA Objects

Sometimes constructing longer messages from LTK-XML files is convenient. For a description of LTK-XML, see Section 3.5. The code below displays an example of constructing an ADD_ROSPEC message from an XML file.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
LTKJAVA

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 40 LTKJAVA – Constructing an LLRP Message from LTK-XML

Note the four exceptions this routine catches. 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.

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 may be present in each parameter or message, see the Octane LLRP reference guide. The example below iterates a custom extension list.

```
LTKJAVA

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

Figure 41 LTKJAVA – Iterating Custom Sub-Parameters

Copyright © 2009, 2010, 2011 Impinj, Inc.

When iterating this list, the application must validate the type of the custom extension before processing the contents and casting the *Custom* parameter back to the appropriate type. An example displays below. 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 42 LTKJAVA – Processing Custom Sub-Parameter

5.3.7 Logging

LTKJAVA uses the de facto standard log4j interface. See http://logging.apache.org/log4j/1.2/index.html for information on using log4j. See the examples below for instances of using log4j in LTKJAVA applications.

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 you 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 compilers 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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
LTKC_INCL = -I$(LTKC_DIR)
CFLAGS = -g $(LTKC_INCL)



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

Figure 43 LTKC – Compiling and Linking

To covert from LTK-XML to LTKC objects, requires libxml2_xxx.a (xxx is either x86, arm, or xScale) as shown below.

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

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

Figure 44 LTKC – Compiling and Linking with Libxml2

LTKCPP -- To add the LTKCPP to your Linux programs, specify the libltkcpp directory in your compilers 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 45 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 below.

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

example1 : example1.c
      $(CCP) $(CPPFLAGS) -o example1 example1.c $(LTKCPP_LIBS) $(LIBXML2_LIB)
```

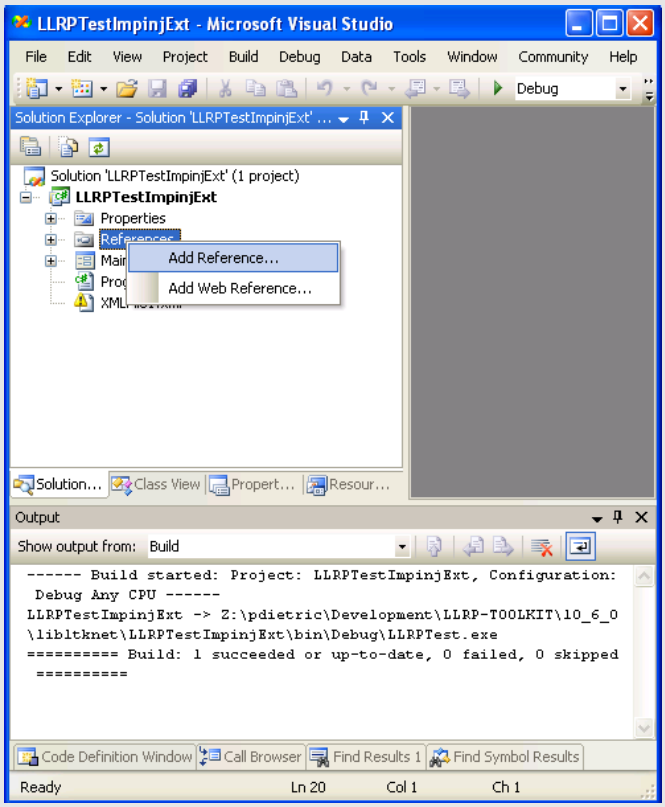

Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 46 LTKCPP – Compiling and Linking with Libxml2

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

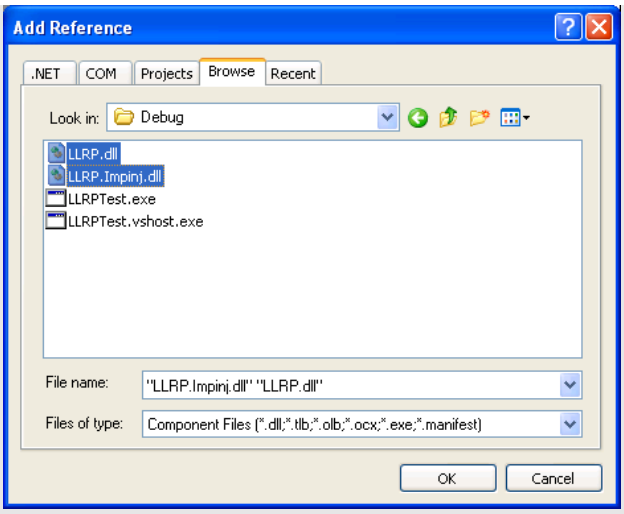
LTKNET

1. Right click references and then select add reference.



2. Select the Browse tab and browse to **`.\libltknet\Debug`** (or Release for a release version).

Copyright © 2009, 2010, 2011 Impinj, Inc.



3. Click OK to add the libraries to your project.

Figure 47 LTKNET – Compiling and Linking

LTKJava –Add the LTKJava to your applications, add the JAR to the classpath. Add the import statements required to use the library. A sample import set is shown below.

```
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 48 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 the Impinj Octane extensions. The following examples display correct initializing of the LTK and register allowing the LTK to send and receive Impinj Octane messages and parameters.

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 49 LTKCPP – Initailizing

```
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 50 LTKNET – Initializing

```
LTKJAVA

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

Figure 51 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 containing the *ConnectionStatusEvent.* The

Copyright © 2009, 2010, 2011 Impinj, Inc.

standard mandates that the client application waits for this notification before continuing Reader communications.

6.3.1 LTKCPP

LTKCPP requires connection objects must be created. The connection object needs a buffer to serialize outgoing and de-serialize incoming messages. Select this buffer based on your application use of LLRP. Choose the max buffer size based on the size of the reports you expect 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. If you are working with immediate reporting (or reports of only a few hundred tags), a 32K buffer size is sufficient.

```
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 52 LTKCPP – Creating Connection

Copyright © 2009, 2010, 2011 Impinj, Inc.

In LTKCPP, the library does not handle the *ConnectionStatusEvent* automatically. Special code requires and verifies that the connection was accepted properly. The example below displays 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;
    }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    /*
     * 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 53 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. Checking the status as well as the command return-code is important.

```
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 that 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 54 LTKNET – Creating/Validating Connection

Copyright © 2009, 2010, 2011 Impinj, Inc.

6.3.3 LTKJAVA

LTKJava awaits the *ConnectionStatusEvent* indicator and throws an exception of the connection fails.

```
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 55 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 ***setMesageID()*** accessor. An LLRP transaction is called and the response is validated. Both messages are freed before completing the operation.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 the response of certain type
     */
    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.
     */
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    return 0;
}
```

Figure 56 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**.** Once created, the **LLRPClient.*CUSTOM_MESSAGE*** is called. If a response is received, it is converted back to the MSG_IMPINJ_ENABLE_EXTENSIONS_RESPONSE via an explicit cast.

```
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 = 1; // This doesn't need to bet set as the library will default

    // 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 57 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*** method**.** Once created, the transaction is performed. If it receives a response, the status code verifies for a successful response.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 58 LTKJAVA – Enabling Impinj Extensions

6.5 Debugging with LTK

All Windows compatible LTK libraries contain accompanying debug versions suitable for common Visual Studio debugging. All Linux compatible LTK libraries supply integrated symbol debugging information. Debug information can be stripped using 'strip' to save program space.

When working 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 display printing 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 structing including messages and parameters.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 59 LTKC – Printing LLRP Messages

6.5.2 LTKCPP

In LTKC, a built in member function ***toXMLString*** for all messages and parameters 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 60 LTKCPP – Printing LLRP Messages

6.5.3 LTKNET

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
LTKNET

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

Figure 61 LTKNET – Printing LLRP Messages

6.5.4 LTKJava

In LTKJAVA, the ***toXMLString()*** method, available for all LLRP messages and parameters, serializes the object as LTKXML. In addition, LTKAVA 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 62 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, and 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 Library
2. Connect to Reader
3. Enable Impinj Extensions
4. 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

8. Process RFID Data (EPC, RSSI, Timestamp)

Sections 6.2 and 0 and 6.3.3 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 exchanges 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>
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
<!—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 63 LTK-XML – Basic Application Sequence

7.2 Setup

Section 4.4 provides detailed Setup information. It is assumed that the programmer has installed:

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

7.3 Factory Default the Reader

Speedway Revolution Readers come with predictable LLRP default values with 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 following examples in the following sections show how this is performed.

7.3.1 LTKCPP

A **CSET_READER_CONFIG** object is created. The related messageID sets via the ***setMesageID()*** accessor, and the FactoryDefault field sets via 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;
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    CSET_READER_CONFIG_RESPONSE *pRsp;

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

    /*
     * Send the message, expect the response of certain type
     */
    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 64 LTKCPP – Factory Default LLRP Configuration

Copyright © 2009, 2010, 2011 Impinj, Inc.

7.3.2 LTKNET

A MSG_SET_READER_CONFIG object is created. Its messageID sets via the MSG_ID property, and the **FactoryDefault** field sets via the `ResetToFactoryDefault` property. The `LLRPClient.SET_READER_CONFIG` is called and the response validates.

```
LTKNET

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

    // factory default 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; //this doesn't need to bet set as 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 65 LTKNET – Factory Default LLRP Configuration

7.3.3 LTKJAVA

A MSG_SET_READER_CONFIG object is created. Its messageID sets via the `setMessageID` method, and the FactoryDefault field is set via the `setResetToFactoryDefault` method. The transact method is called and the response validates.

```
LTKJAVA
    private void factoryDefault() {
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        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 66 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 controls the Reader start and stop manually, 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 it is recommended this field is set to zero.

LLRP *ROSpec* does not support Priority in Speedway Revolution 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 with no *AISpec* specific stop conditions. To specify all antennas, LLRP reserves the value of zero.

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, STOP_ROSPEC messages, and when reporting the ROSpecID in the *TagReportData* parameters. The application assigns a number and

Copyright © 2009, 2010, 2011 Impinj, Inc.

is only for identification purposes. Using a number significant to your application is recommended. LTK and LLRP allow any 32-bit number (except 0) for these IDs.
The LTK-XML representation of this simple *ROSpec* is shown below:

```
LTKXML

<?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 67 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 explicit setting. Following this, the parameter is inserted into its parent parameter or message. As an example (below), the **`CROSpecStartTrigger`** and **`CROSpecStopTrigger`** are created and then inserted into the newly created **`CROBoundarySpec`**. This procedure repeats for each of the parameters within the *ROSpec*.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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

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

Constructing *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;

    /*
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
     * 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 the response of certain type
     */
    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;
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
}
```

Figure 68 LTKCPP – Constructing a Simple ROSpec

7.4.2 LTKNET

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

- programmatically with LTKNET C# objects
- via importing LTK-XML descriptions

For simple messages (see 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 it you desire "tuning" 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 may be easier using the programmatic approach. The object based construction display below. For an example of the LTK-XML based construction see Section8.5.1.

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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)
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

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

Figure 69 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
- via importing LTK-XML descriptions

For simple messages (see 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 it is desirable 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 may be easier to use. An object based construction is shown below. 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));
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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 70 LTKJava – Constructing a Simple ROSpec with Objects

7.5 Enable an ROSpec

A *ROSpecs* must be added in the Disabled state: the *ROSpec* must be enabled before responds to a START_ROSPEC message. The samples below follow the same pattern as the factory default described in Section 7.3. Note, when enabling and starting *ROSpecs*, it is important to use the ROSpecID matching the ID from the *ROSpec* added in Section 7.4.

```
LTKCPP
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
{
    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 the response of certain type
     */
    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;
}
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 71 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 72 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");
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        }
        else {
            logger.error(response.toXMLString());
            logger.info("ENABLE_ROSPEC_RESPONSE failed ");
            System.exit(1);
        }
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

Figure 73 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 using the START_ROSPEC and STOP_ROSPEC messages respectively. In applications using internal Reader triggering, this step is not necessary. These samples 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 matching the ID from the *ROSpec* 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 the response of certain type
     */
    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;
    }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    /*
     * 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 74 LTKCPP – Start an ROSpec

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 75 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);
        }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

Figure 76 LTKJava – Start an ROSpec

7.7 Handling Tag Report Data

Data returns to the Reader via LLRP RO_ACCCESS_REPO**RT** messages. The handling of these messages is notably different in the different LTK implementations.

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/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 is shown below in LTKCPP:

```
LTKCPP

        pMessage = recvMessage(1000);

        /* validate the timestamp */
        tempTime = time(NULL);
        if(difftime(tempTime, startTime) > timeout)
        {
            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;
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            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
            {
                /*
                 * This should never happen. Using continue
                 * to keep indent depth down.
                 */
                printf("WARNING: READER_EVENT_NOTIFICATION without data\n");
            }
        }
```

Figure 77 LTKCPP –Receiving Asyncrhonous Reader Messages

Tag reports return in a list. To access an individual report, the list can be iterated with a standard STL iterator as follows.

```
{
    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 78 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:

Copyright © 2009, 2010, 2011 Impinj, Inc.

- RO_ACCESS_REPORT – These are the tag reports from both inventory and access operations. Reports deliver through 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 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 validating connectivity. The Reader itself does not take action when these events are not delivered. These messages are delivered through delegate **delegateKeepAlive**.

The following example shows installs 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 79 LTKNET –Adding Event Handlers for Asyncrhonous Reader Messages

The user defined methods calls for each message. In our simple example, we to 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 below. Note that in general more than one tag report can come in each message even with the default immediate reporting setting.

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 80 LTKNET –Receiving Asyncrhonous 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 and dispatches 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 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. Impinj recommends 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) {
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        // convert all messages received to LTK-XML representation
        // and print them to the console

        logger.debug("Received " + message.getName() + " message asychronously");

        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 81 LTKJava –Receiving Asyncrhonous 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();
        }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        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);
        //logger.debug(output);
    }
```

Figure 82 LTKJava –Extracting Data From Tag Reports

8 Impinj Octane LTK Application

This section provides access to 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

An application wishes to configure a two-antenna portal to track items flowing through the portal. To achieve peak performance, the system engineer was to 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 A.C. power or interfering with other RFID devices when no tags are present. The system is required to work on both FCC and ETSI Speedway Revolution readers.

8.1 Analysis

The Speedway Revolution Readers default to 30 dBm transmits power (typically) which is dependent on the Regulatory region. Some versions of Speedway Revolution 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. One could search the data sheet for Speedway Revolution Readers in FCC and ETSI and hard code the LLRP power index via a lookup table based on region and model #. Alternately one could query the Reader for the 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 containing this information. The LLRP *RFTransmitter* parameter sets the transmit power. However, is also contains the hopTableID and ChannelIndex. They cannot be set independently, the application must query these three items for use in building the *RFTransmitter* 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

Implementing the application flow use case requires the following steps: the:

1. Initialize the Library.
2. Connect to the Reader.
3. Enable Impinj Extensions.
4. Use the Factory Default LLRP configuration to ensure the Reader is in a known state. (We are relying on the default Reader configuration for this simple example.)
5. GET_READER_CAPABILITIES to learn the maximum power supported by this Reader.
6. GET_READER_CONFIG to get the hopTableID and channelIndex for use later when setting the *RFTransmitter* parameter.
7. SET_READER_CONFIG with the appropriate *InventorySearchMode*, *TagDirection*, *LowDutyCycleMode*, *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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

11. Process RFID Data (EPC, RSSI, Timestamp) and direction data.

We have seen steps 1-4, 9 and 10 in this document. We will begin our example 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_REPSONSE 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 report in a table within the *UHFBandCapabilities* parameter which is within the GET_READER_CAPABILITIES_REPONSE 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. To validate this parameter, since it may be used differently by different vendors, the application needs to check the DeviceManufacturerName and modelName combination. 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; // not this doesn't need to bet set as 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;
    }
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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 revolution.
    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 83 LTKNET – Retreiving 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;
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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 the response of certain type
     */
    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;
    }

    /*
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
** 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.
 */
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    return 0;
}
```

Figure 84 LTKCPP – Retreiving Device Capabilities

In LTKJava, the power level entries are stored in a standard Java List object. The code below shows a sample 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 addition 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();
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
                // 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 85 LTKJAVA – Retreiving Device Capabilities

8.4 Get Reader Configuration

As mentioned above, to set the *RFTransmitter* parameter which contains the transmit power level, it also requires setting the 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 device. Because Speedway Revolution Readers default channel to the same values 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; // not this doesn't need to bet set as 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());
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            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);
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
/*
 * Send the message, expect the response of certain type
 */
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",
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            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 {
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
                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 below 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>
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
                <!--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 86 LTKXML – Impinj Octane LLRP Extensions

8.5.2 LTKNET

The C# code provisions this LTKNET configuration on the Reader displays below. 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;
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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());
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

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

Figure 87 LTKNET – Setting Configuration with LTK-XML

8.5.3 LTKCPP

To set advanced configuration within a **CSET_READER_CONFIG**, the example 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++);
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

Figure 88 LTKCPP – Importing SET_READER_CONFIG from LTK-XML File

Modifying the transmit power requires the application must access the **`CAntennaConfiguration`** objects within this message, and then create or modify the **`CRFTransmitter`** object and set the power level index to the value determined in step 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 the response of certain type
 */
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;
}

/*
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
     * 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 89 LTKCPP – Setting Antenna Configuration and Transmit Power

8.5.4 LTKJAVA

The Java code to provision the configuration displays below. Note specifically how the XML reads (and exceptions handled). In addition, notice how to access the parameter fields, and finally, 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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        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 90 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 wishes 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 and accurate correlation between them.

Copyright © 2009, 2010, 2011 Impinj, Inc.

Two types of tags will pass through the portal: assets and personnel tags. For asset tags, all the information is contained within the EPC. For personnel tags, however, the Gen2 user memory contains a code describing 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 excepting GID-96 and GRAI-96.

Because the Reader should minimize its RF interference, use only low duty cycle mode as 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 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 to for RFID data and Process RFID Data (EPC, and Timestamp).

Steps 1-5, 7, 9, and 11 were covered in earlier examples. Examples for steps 6, 8 and 12 follow in this section.

9.3 Report Accumulation

The *ROReportSpec* contains the rules defining 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 *tagReportContentSelector*. This requires an accurate association of

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 poll 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 to more use cases and because it re-uses the ROSpec from previous examples.*

9.3.1 LTKNET

In LTKNET, the sample uses a similar SET_READER_CONFIG message as **`docsample2`** (Section8.5.2), along with the same supporting code to provision the Reader settings except 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>
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            <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 91 LTKXML – SET_READER_CONFIG for docSample3

9.3.2 LTKCPP

In LTKCPP, the sample below 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);
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    pROrs->setN(0);

    /*
    ** 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);
    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 the response of certain type
     */
    pRspMsg = transact(pCmd);

}
```

Figure 92 LTKCPP – SET_READER_CONFIG for docSample3

9.3.3 LTKJAVA

In LTKJAVA, the sample uses a similar SET_READER_CONFIG message as **`docsample2`**, along with the same supporting code to provision these settings on the Reader except 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 non GID-96 and GRAI-96 codes. Because the system must preserve link bandwidth, we add *C1G2Filters* to the Reader which filters out tags not matching 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 on 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" on reading the EPC header. However, the PC bits also contain several other indicators (UMI, XPC etc) that are*

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 not matching the filter (GRAI) and includes (selects) all tags matching the filter. The second filter uses the Select_DoNothing action which builds on the first filter by adding those tags matching 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 93 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 93 LTKNET – InventoryFilters for docSample3

This *AntennaConfiguration* element is included in the *ROSpec* and added using code that is similar to Section8.5.2.

9.4.2 LTKCPP

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

Copyright © 2009, 2010, 2011 Impinj, Inc.

NOTE: the *C1G2TagInventoryMask* specified pointers and lengths 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();
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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 the *ROSpec*. Figure 93 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 94 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

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 the EPCglobal EPC header corresponding 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 matching 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 tags this *AccessSpec* applies to. The second in the Read command 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. It is recommended to set 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 an *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 display below and provide a comparison so you can choose the best method for your application. Figure 95 shows the LTK-XML for the *AccessSpec* required for this use case. Figure 96 shows the C# code required to import and send this *AccessSpec* to the Reader. Figure 97 shows the addition of the same *AccessSpec* using C# objects to construct the various parameters and fields. Note the TagMask and TagData which mark the 11 C1G2PC bits as "don't care".

```
LTK-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>
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        <!--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 95 LTKXML –AccessSpec to read Tag User Memory

The C# code to pass this command to the Reader is similar to 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)
        {
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            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 96 LTKNET –Adding an AccessSpec with LTK-XML

Adding the same *AccessSpec* (except for a unique ID) using C# programming objects rather than XML displays below.

```
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];
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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();
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
            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 97 LTKNET –Adding an AccessSpec with C# Objects

Processing RO_ACCESS_REPORTS containing *OpSpecResults* is similar to what we saw in Section7. One notable difference:

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();
            }
        }
    }
}
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
#endregion
```

Figure 98 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();
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    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 the response of certain type
     */
```

Figure 99 LTKCPP –Creating an AccessSpec

Processing RO_ACCESS_REPORTS containing *OpSpecResult*s is similar to what we saw in Section 7. One notable difference:

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
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    */
    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 100 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.

```
    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 SET_READER_CONFIG message

            response = connection.transact(addAccessMsg, 10000);

            // check whetherSET_READER_CONFIG 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);
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        } 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 101 LTKJAVA –Creating an AccessSpec

Processing the results in the tag reports, we use standard list iteration on the **opSpecResultList**. Below shows a snippet of code that only looks for the C1G2ReadResult 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 102 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 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 generates a RO_ACCESSS_REPORT message, clear the internal database and begin 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 Speedway Reader always generates a* 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 103 LTKNET –Trigger Asyncrhonous 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 104 LTKCPP –Trigger Asyncrhonous 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

private void getReport() {
    GET_REPORT grep = new GET_REPORT();
    grep.setMessageID(getUniqueMessageID());
    logger.info("Sending GET_REPORT ...");
    connection.send(grep);
}
```

Figure 105 LTKJAVA –Trigger Asyncrhonous reports with GET_REPORT

Copyright © 2009, 2010, 2011 Impinj, Inc.

10 Impinj Octane LTK Tag Data

This section shows how to access the Octane Impinj LLRP custom tag data elements via the Impinj LTK. 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 wishes to get more low-level data regarding tags as they pass through a portal. The application wants to use this data to estimate the instantaneous velocity of the tag to help determine direction and timing through the portal.

10.1 Analysis

To use the low-level data available via Speedway Revolution, the application must configure the tag data reported via LLRP. Some of the data is available via 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), and in this case DRM M=4 is chosen for the LTKCPP example and Max throughput is chosen for the LTKNET example.

Once 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 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, this discussion and example begins with step 5. For the complete sample code, see the **docSample4** directories within the Impinj LTK distribution.

Examples for both LTKCPP and LTKNET show the object based parameter and message construction below.

Copyright © 2009, 2010, 2011 Impinj, Inc.

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 with 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 enables the required data for our velocity estimate. Note: Setting the session, *ImpinjInventorySearchMode* and ModeIndex valuesin 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>
            <Impinj:ImpinjInventorySearchMode
                 xmlns="http://developer.impinj.com/ltk/schema/encoding/xml/1.8">
                <!--Need Dual-target for lots of low level data -->
                <InventorySearchMode>Dual_Target</InventorySearchMode>
            </Impinj:ImpinjInventorySearchMode>
            <Impinj:ImpinjLowDutyCycle
                xmlns="http://developer.impinj.com/ltk/schema/encoding/xml/1.8">
                <LowDutyCycleMode>Disabled</LowDutyCycleMode>
                <EmptyFieldTimeout>10000</EmptyFieldTimeout>
                <FieldPingInterval>200</FieldPingInterval>
            </Impinj:ImpinjLowDutyCycle>
        </C1G2InventoryCommand>
    </AntennaConfiguration>
    <ROReportSpec>
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
        <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 106 LTKXML –Setting Tag Report Contents

10.3.2 LTKNET

Setting the configuration in LTKNET via XML is identical to the previous examples.

10.3.3 LTKCPP

In LTKCPP, we use the object based construction. However, LTK-XML construction is also possible using the LTKCPP LTK-XML examples shown above. Previous examples show the SET_READER_CONFIG based construction in LTKCPP. The example below provides only the section of code that adds the `CTagReportContentSelector`. **`docsample4.cpp`**. provides 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);
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
/* 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 107 LTKCPP – Setting Tag Report Contents

10.3.4 LTKJAVA

Setting the configuration in LTKJAVA via XML 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 78 shows how to get an individual tag report from the RO_ACCESS_REPORT message. Once this report is extracted, the application must look inside to find the low-level data.

In the LTKCPP, custom parameters are stored as STL lists. For tag information set through the *ImpinjTagReportContentSelector*, the data is reports via the *CustomParameter* within the standard

Copyright © 2009, 2010, 2011 Impinj, Inc.

LLRP *TagReportDataParameter*. Note the type of each custom parameter is checked before calling the function 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 108 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 109 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`. Code below shows a snippet from the tag data processing function.

```
LTKJAVA

List<Custom> clist = tr.getCustomList();
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
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 referenced in Section 1.6.8. Please see the source code samples in **`docsample4.cpp`** and **`docsample4.cs`**, and **`docsample4.java`**.

11 Using Monza4 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 M4 QT feature set see the reference in Section1.6. The only new programming exercise to use M4, aside from understanding the functionality, utilizes custom *AccessSpecs* via the *CustomParameter* in LLRP standard.

11.1 Adding Custom OpSpecs

Add Custom OpSpecs to the standard list of OpSpecs using the complete example shown in Section 9.

11.1.1 LTKCPP

LTKCPPuses the ***`addAccessCommandOpSpec`*** method to add a custom standard OpSpec to the **`AccessCommand`** object.

```
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 */
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
    pAccessCommand->addAccessCommandOpSpec(psetQT);
```

Figure 110 LTKCPP – Adding Custom OpSpec

11.1.2 LTKNET

LTKNET uses the ***Add*** method of the **AccessCommandOpSpec** class to add a custom OpSpec.

```
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 111 LTKNET – Adding Custom OpSpec

11.2 Retrieving Custom OpSpecResults

11.2.1 LTKCPP

In LTKCPP, *OpSpecResults* store as STL lists. The application must iterate the list and examine each type of result, and then act appropriately.

```
LTKCPP
    for (
        OpSpecResults = pTagReportData->beginAccessCommandOpSpecResult();
        OpSpecResults != pTagReportData->endAccessCommandOpSpecResult();
        OpSpecResults++)
        {
            else if( (*OpSpecResults)->m_pType ==
                        &CImpinjSetQTConfigOpSpecResult::s_typeDescriptor)
            {
                written = formatOneSetQTConfigResult(*OpSpecResults,
                                                ptr, bufLimit, "\n    SETQT ");
                ptr += written;
                bufLimit-=written;
            }
    }
```

Figure 112 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.

```
LTKNET
if ((msg.TagReportData[i].AccessCommandOpSpecResult != null))
{
   for(int x = 0; x <
         msg.TagReportData[i].AccessCommandOpSpecResult.Count; x++)
```


Copyright © 2009, 2010, 2011 Impinj, Inc.

```
   {
        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 113 LTKNET – Retreiving Custom OpSpecResults

Copyright © 2009, 2010, 2011 Impinj, Inc.

Appendix A: Glossary

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

AISpec: Antenna Inventory Specification. A *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 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 to allow 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 leads the development of industry driven standards for the Electronic Product Code (EPC) to 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, extended, compiled, tested, and packaged for use with Speedway Revolution Readers by Impinj.

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 is a data element 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.

Copyright © 2009, 2010, 2011 Impinj, Inc.

12 Revision History

Date	Revision	Comments
09/01/2009	10.8.0	Initial release
03/14/2010	10.10.0	Update for addition of LTK-XML to LTKCPP
10/28/2010	10.12.0	Update for Octane 4.6
4/25/2011	10.14.0	Add support for LTKJAVA

Copyright © 2009, 2010, 2011 Impinj, Inc.

Notices:

Copyright © 2009,2010, 2011 Impinj, Inc. All rights reserved.

The information contained in this document is confidential and proprietary to Impinj, Inc. This document is conditionally issued, and neither receipt nor possession hereof confers or transfers any right in, or license to, use the subject matter of any drawings, design, or technical information contained herein, nor any right to reproduce or disclose any part of the contents hereof, without the prior written consent of Impinj and the authorized recipient hereof.

Impinj reserves the right to change its products and services at any time without notice.

Impinj assumes no responsibility for customer product design or for infringement of patents and/or the rights of third parties, which may result from assistance provided by Impinj. No representation of warranty is given and no liability is assumed by Impinj with respect to accuracy or use of such information.

These products are not designed for use in life support appliances, devices, or systems where malfunction can reasonably be expected to result in personal injury.

The "Powered by Impinj" shield is your assurance of RFID integrity

Copyright © 2009, 2010, 2011 Impinj, Inc.