RFTagAware™

Programmer Guide

Version 1.3

June 2005

ConnecTerra, Inc.

100 CambridgePark Drive

Cambridge, Massachusetts 02140

Phone: (617) 441-2200

Facsimile: (617) 492-5837

Web Site: www.connecterra.com

Email: info@connecterra.com

ConnecTerra® is a leading provider of enterprise software for device computing.

Contents

Preface

Purpose of This Manual

This manual describes the Application Level Events (ALE) API. This API lets applications use the RFTagAware Edge Server to access information from RFID tag readers.

Audience

- IT procurement specialists.
- Software engineers and other developers using the Application Level Events (ALE) API.
- Network architects and engineers.
- Technology strategists and senior managers.

Users should know how to program in Java™.

Related Documents

- *RFTagAware Deployment Guide*
- *RFTagAware Reader Configuration Guide*

What's in This Manual

- Chapter 1: Introduction

  This chapter provides an overview of RFTagAware.

- Chapter 2: Reading and Writing Tags

  This chapter describes how to use RFTagAware to read and write tags.

- Chapter 3: Asynchronous Notification Mechanisms

  This chapter describes the asynchronous notification mechanisms RFTagAware uses to deliver reports.

- Chapter 4: Triggers

  This chapter describes trigger mechanisms that start and end event cycles.

- Chapter 5: Reading Tags Using the ALE API

This chapter describes the ALE API programming components you use to read tags.

- Chapter 6: Writing Tags Using the ALE API

  This chapter describes the ALEPC API programming components you use to write tags.

- Chapter 7: Sample Java Applications

  This chapter walks you through sample Java applications that use the ALE and ALEPPC APIs.

- Chapter 8: Sample .NET Applications

  This chapter walks you through sample .NET applications that use the ALE and ALEPC APIs.

Contacting Technical Support

For technical support, call:

617-441-2280

Monday-Friday 9am-5:30pm Eastern Time.

ConnecTerra, Inc.
100 CambridgePark Drive
Cambridge MA 02140

Voice: +1-617-441-2200
FAX: +1-617-492-5837

support@connecterra.com

Chapter 1: Introduction

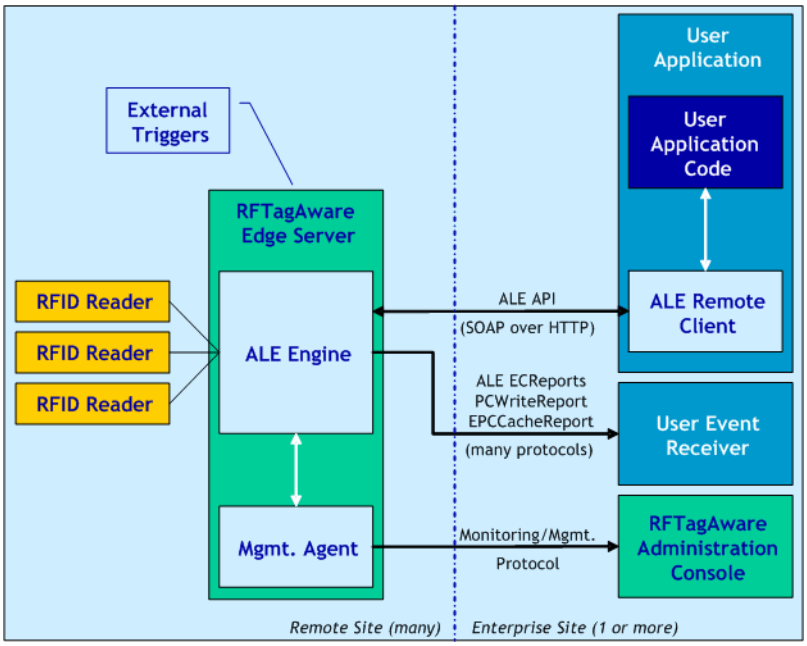
Contents

This chapter describes the RFTagAware architecture and its Application Programming Interface (API).

RFTagAware Architecture

RFTagAware software has two main components:

- Edge Server (page 1-2)
- Administration Console (page 1-2).



Edge Server

The **Edge Server** is software deployed at a remote site where there are RFID readers.

There are two main subcomponents within the Edge Server:

- The first is an Application Level Events (ALE) processing engine. The ALE Engine is responsible for receiving and processing RFID tag data on behalf of user applications.
- The second is a monitoring and management agent that allows the health and functioning of readers and Edge Server software to be monitored and managed remotely through the RFTagAware Administration Console.

Administration Console

The Administration Console is software typically deployed at a centralized enterprise site, where operations staff may use it to monitor and manage the RFID infrastructure. The Administration Console is also useful for on-site tuning and debugging. The Administration Console includes an interactive user interface for all of these operations.

Standards Compliance

RFTagAware is compliant with all relevant standards of EPCglobal. ConnecTerra is committed to revising RFTagAware so that it is always in compliance with these standards.

RFTagAware conforms to EPCglobal standards in the following way:

- RFTagAware works with RFID readers that implement the following protocols:
  - EPCglobal Class 0, Class 0+, and Class 1 RF Protocols.
  - ISO 15693 RF Interface protocol.
  - ISO 18000-6B RF Interface protocol.

  RFTagAware will also work with RFID readers that implement the forthcoming EPCglobal UHF Generation 2 RF protocols, when such readers become available.
- RFTagAware supports the EPCglobal Tag Data Standard. This standard governs the bit-level encoding of object identity and other information onto RFID tags. It also specifies a URI-based syntax for exchange of tag data between software application components, and a second URI-based syntax for the description of filtering patterns. RFTagAware fully implements all of these standards.
- RFTagAware includes components that play the filtering and collecting role as defined within the EPCglobal Architecture Framework. RFTagAware fully implements the EPCglobal Application Level Events (ALE) specification, which is the standard interface to filtering and collection functionality as defined by EPCglobal (replacing earlier "Savant" specifications). RFTagAware also provides other value add functionality in the areas of reader management, application integration, and tag writing.

  The EPCglobal Application Level Events (ALE) API was developed by ConnecTerra and other EPCglobal member companies under the auspices of the EPCglobal Software Action Group. ALE is in the final stages of standardized within EPCglobal. ConnecTerra is committed to the continued evolution of ALE as a standard.

The ALE API

This section describes the Application Level Events (ALE) application programming interface (API).

Overview

Applications interact with the Edge Server through an application programming interface (API) called Application Level Events, or ALE. ALE provides a high-level, declarative way for applications to read and write RFID data, without requiring application programmers to interact directly with RFID readers or perform any low-level real-time processing or scheduling operations. ALE therefore logically occupies a position between application business logic and low-level RFID tag reads and tag writes, providing a strong degree of insulation between the two.

In RFTagAware, ALE processing takes place within the Edge Server, so that the large volumes of RFID read data can be reduced to interesting business events prior to having to travel over local or wide-area enterprise networks to applications.

Basic ALE Operation

The basic concept of ALE is quite simple. An application makes a request to the ALE interface to write or read tags. The ALE engine within the RFTagAware Edge Server processes the data from the readers, and generates reports back to the application based on the conditions specified in the request.

A tag reading request may be a one-time request that is satisfied synchronously ("give me a list of the EPC codes that are currently stable in the field of readers at Dock Door 5") or may be a standing request that generates asynchronous notifications when events of interest happen ("every ten minutes tell me how many new things have shown up").

A tag writing request may also be satisfied synchronously ("write the following tag now") or may be a standing request that writes a tag and generates a report when some external event happens ("when the case crosses the electric eye beam, write the tag").

Benefits

The ALE API as implemented in ConnecTerra's RFTagAware provides a number of unique benefits:

- Rapid development and deployment — different teams can independently get new projects quickly up and running with a minimum amount of programming.
- Flexibility to adapt to changes quickly — individual applications can be easily modified while the Administration Console enables rapid reconfiguration of the overall deployment.
- Variety of deployment scenarios — enterprise applications can be distributed across remote sites using a wide range of network transports and protocols.
- Built-in support for multiple applications sharing readers — the Edge Server includes logic and security to handle multiple independent applications simultaneously.
- Manageability, security and integrity — centralized management of the control of the overall deployment supports use of RFID data in enterprise applications.
- Scalability of readers, applications, servers and sites — scalable number of readers per RFTagAware, number of RFTagAware instances per site, and number of sites in the overall application. Designed with future readers in mind.
- Standards leadership and support.
- RFTagAware's implementation of the ALE API includes RFTagAware-specific extensions. RFTagAware extensions are explicitly noted throughout this document.

Overall these benefits mean that simple applications are developed and deployed quickly, the solution scales gracefully with new readers and applications, and the resulting solution meets enterprise requirements for business critical infrastructure.

Programming Languages

To use ALE from a program, use one of the following methods:

- You can use standard SOAP-based web services development tools to generate an ALE client stub for any programming language. The RFTagAware installation provides WSDL files that you may use for this purpose. The WSDL files are in your RFTagAware installation directory at:
  `/share/schemas/EPCglobal-ale-1_0.wsdl`
  `/share/schemas/ALEPCService.wsdl`
- RFTagAware provides a Remote Client library for the Java programming language, which Java programs may use to access an RFTagAware Edge Server via the ALE API.

Directory Structure Concepts

Defaults and Allowed Install Locations

RFTagAware's default installation directories are:

- Windows: `C:\Program Files\ConnecTerra\RFTagAware`
- UNIX: `/opt/ConnecTerra/RFTagAware`

You can choose different primary install locations.

Directory Structure

Regardless of where you put your primary install directory, RFTagAware creates two main subdirectories beneath the primary install directory:

- `control`

  Version-independent location for the master startup scripts for RFTagAware's various components. These scripts act like pointers or symbolic links to the actual scripts for a particular version. This lets you continue to use the same `/control/bin/script_name` path to run a script, no matter how many times you upgrade.

- `version_number`, for example: `1.3.0`

  Contains the files for a specific version of RFTagAware.

Directory Tree Overview

Here are some highlights from a sample installed directory tree:

`control`	

`control/bin`	Use these master scripts to start the Administration Console, Edge Server, Reader Simulator, and Quick Test utility:
	`RunAdminConsole`
	`RunEdgeServer`
	`RunReaderSim`
	`RunQuickTest`
	These scripts will always invoke the version of RFTagAware that you have most recently installed.

1.3	
1.3/bin	Version-specific scripts used by RFTagAware. Use these scripts to start a *particular* version of RFTagAware, even if it is not the version you most recently installed.
1.3/etc	Properties and logging files for the Administration Console, Edge Server, and JMS Notification Driver: admin-console.props edge.props jms.options logging.props.admin-console logging.props.edge
1.3/lib	Java libraries used by RFTagAware components.
1.3/samples	Sample Java source code that illustrates how to program the ALE API.
1.3/share/schemas	Schema used to represent RFTagAware data types in XML and WSDL files describing the ALE API.
1.3/UninstallerData	Files used by the installer during uninstallation.
1.3/var	Files created and used by RFTagAware during operation, including log files. The edgestate subdirectory contains state data about ECSpec, PCSpec, and EPCCacheSpec instances and their subscribers, and reader configuration data for readers configured using the Administration Console, as well as other persistent data that you create using the ALE API.

Shortened Pathnames in Documentation

RFTagAware documentation refers to the RFTagAware subdirectories (for example, bin and etc) directly, without the preceding full pathname. In all cases, these shortened pathnames refer to subdirectories within the default or user-specified RFTagAware installation directory.

Chapter 2: Reading and Writing Tags

Contents

This chapter describes how to use RFTagAware to read and write tags.

Application Level Events (ALE)

The basic concept of ALE is quite simple. An application makes a request to the ALE interface to write or read tags. The ALE engine within the RFTagAware Edge Server processes the data from the readers, and generates reports back to the application based on the conditions specified in the request. For more information on ALE, see Basic ALE Operation on page 1-4

The remainder of this chapter provides a functional overview of ALE tag reading and writing.

Reading Tag Data

To read tag data, applications define *event cycle specifications* (`ECSpec`), which specify what RFID data is of interest. An example of the content of an `ECSpec` is "give me a report every 60 seconds of what objects have been added or removed to warehouse shelves #4 and #5, including Acme products only and excluding pallet-level data."

Once an application defines an `ECSpec`, the application receives RFID data through *event cycle reports* (`ECReports`). In the previous example, the ALE engine within the RFTagAware Edge Server would generate a new `ECReports` instance every 60 seconds, containing a list of objects added or removed from the warehouse shelves as specified.

For a detailed overview of `ECSpec` and `ECReports`, see ECSpec on page 5-6 and ECReports on page 5-17.

An application may define and subscribe to an `ECSpec` programmatically by using the ALE Remote Client to access the ALE API directly. Through the API, user application code may define an `ECSpec` and subscribe one or more destinations to an `ECSpec` for asynchronous delivery of `ECReports`. The API also permits user application code to request the delivery of an `ECReports` instance on demand, in a synchronous manner.

Read Cycles and Event Cycles

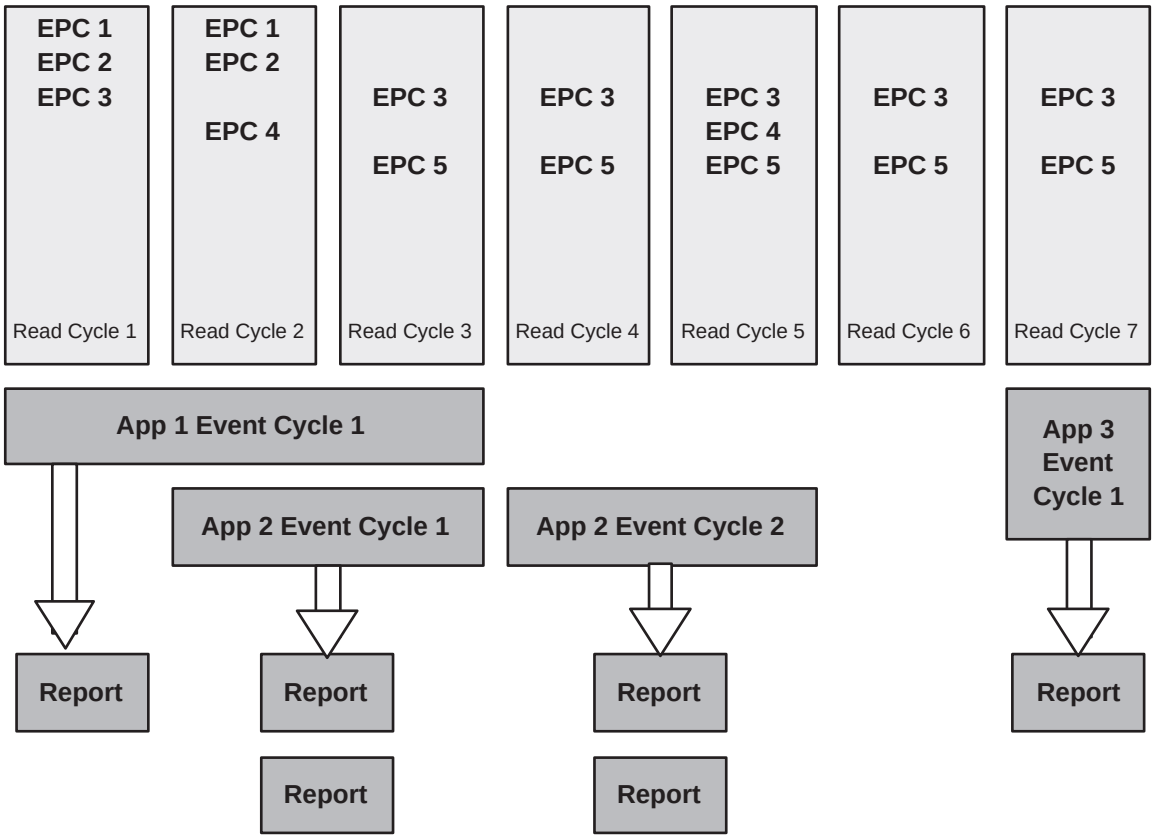
RFID readers generally scan for tags much more frequently than real-world applications require data. In addition, the likelihood of an RFID reader actually reading a tag during any one attempt is sensitive to a large number of factors, including the position and motion of tags, presence of objects or people, and even the activity of other readers. Because of this, applications generally use the data accumulated from a number of RFID reads.

In recognition of this, the ALE interface distinguishes between the rate at which readers scan for tags from the rate at which applications receive data. A Read Cycle refers to a single complete scan of all tags in a single reader antenna's field. This generally happens a few times a second. An Event Cycle is one or more read cycles, from one or more readers, that are to be treated as a unit from an application

perspective. The data of an event cycle consists of all the tags seen in any of the read cycles by any of the readers.

At the completion of an event cycle, the ALE engine within the RFTagAware Edge Server processes the set of tags the readers saw during that cycle and generates one or more reports to the requesting application. Each report specification may include different criteria for reporting, such as whether to report all tags or only changes, whether to include actual tag IDs or just counts, whether to include or exclude certain tags based on their identities, and how filtered EPCs are grouped together for reporting. Because different applications may be interacting with a single Edge Server, there may be many overlapping event cycles in progress at any one time, sharing the data from overlapping sets of readers in arbitrary ways.

The following picture shows the relationship of read cycles, event cycles and reports.



There are a number of ways to specify the boundaries of event cycles relative to read cycles:

- Time: the event cycle is specified to last a certain amount of time, independent of how many read cycles are involved.
- Number of read cycles: the event cycle is specified to last a fixed number of read cycles.

- Field stability: the event cycle is specified to complete when the set of tags read by a reader has been stable for a specified period of time.
- External events: the event cycle is specified to begin or end when an external event happens, such as a container passing an electric eye beam, or a human pressing a button.

Multiple end conditions may be specified, with the event cycle concluding when any of the conditions are met. An example may be to specify an event cycle to last 10 seconds or if the field is stable for at least 5 seconds.

Smoothing Read Cycles with Transient Filtering

RFTagAware provides a transient filtering mechanism that can be used to help produce smoother results when tags are not read reliably by readers. You can use the transient filter when you wish to filter out tags that appear only briefly, keeping those tags that are read several times within a specified interval of time. The transient filter may also be used to smooth over gaps when tags disappear briefly (though accumulation of multiple read cycles into an event cycle has a similar effect, even without transient filtering).

For instructions on adding a transient filter, see Adding a Transient Filter on page 2-13.

Interacting with ALE

Applications interact with the ALE engine through event cycle specifications (`ECSpec`) and event cycle reports (`ECReports`). An `ECSpec` identifies the specific information or events that an application is looking for in each event cycle. The `ECSpec` defines what locations (logical readers) are to be included, what external events or time parameters define the start and stop of an event cycle, and a set of report specifications, each defining a subset of the data of interest. RFTagAware's ALE engine can process large numbers of `ECSpec` instances from different applications simultaneously.

There are three modes of interaction between an application and the RFTagAware Edge Server:

- **Immediate**: The application uses the ALE interface to send an `ECSpec`, and the ALE engine within the Edge Server fulfills the specification by completing one event cycle, after which the application receives the corresponding reports.
- **Immediate with predefined request ("poll")**: Applications can request a single event cycle from a previously defined `ECSpec` and receive the reports in the response.
- **Asynchronous ("subscribe")**: An application subscribes to a previously defined `ECSpec` using the `subscribe` operation, indicating an address to which reports should be delivered. The ALE engine within the Edge Server then sends reports to the application as each event cycle completes, continuing to do so until the application cancels the subscription.

An application or user can define a standing `ECSpec` at any time. The Edge Server remembers all such `ECSpec` instances until an application or the user explicitly undefines them; thus, numerous applications can poll or subscribe to the same `ECSpec`.

The Edge Server remembers all subscribers until they unsubscribe. Thus, if the application subscribes and then exits, the Edge Server will continue to send reports.

The `immediate` and `poll` methods are synchronous, insofar as the application blocks after making its request until the ALE engine responds with the corresponding reports. The `subscribe` method is asynchronous, as control is returned to the application immediately after processing the `subscribe` call, with subsequent reports being delivered via an asynchronous channel.

Reports

An `ECSpec` specifies one or more reports that may be generated at the end of each event cycle. The report is based on the complete list of tags that were detected by the specified readers during the event cycle. Each tag that was read appears once in the list, even if it was detected in multiple read cycles on one reader or on multiple readers. Based on this list of tags, each report has a number of options:

1. What tags should be included for consideration: all tags read during this event cycle, only those tags that are new relative to the last event cycle for the same request (additions), or only those tags that were present during last event cycle for the same request but which no longer are present (deletions). Note that the latter two choices do not apply to a one-time, immediate request as there is no previous event cycle to compare against.
2. What filters should be applied to the list of tags from Step 1. A filter may specify that certain tags should be excluded from consideration ("do not include any Acme products"), or that only certain tags should be included ("only include pallet-level tags"). EPC Patterns on page 5-13 includes a detailed description of filter options.
3. If there are no tags left after Step 2, whether a report should be generated or not.
4. How filtered EPCs are grouped together for reporting.
5. When a report is generated, should the report enumerate the actual tag identities that result from Steps 1, 2, and 4, or merely include a count of the number of tags left in each group after Steps 1, 2, and 4.

The option in Step 3 is useful for specifying that applications receive reports *only* for event cycles where something of interest actually occurred. For example, in a warehouse application that is monitoring what goods are present on a shelf, the option in Step 3 may be used so that the ALE engine sends a report only when something is placed on or removed from the shelf.

For more information on reports, see Chapter 5: Reading Tags Using the ALE API.

Writing Tag Data

The process of instructing a reader to encode an EPC value onto an RFID tag is called both "writing" and "programming." Tag writing can be performed both by RFID readers, and by RFID-enabled printers, which can print labels with embedded tags. For a complete list of the readers and printers for which RFTagAware supports tag writing, as well as the specific tag formats which are supported for each device, see the supported RFID readers section of the *RFTagAware Reader Configuration Guide*.

To write tag data, applications define *programming cycle specifications* (`PCSpec`), which specify an interval of time during which a single tag is written and verified. At the end of a programming cycle, applications receive a write report (`PCWriteReport`), which tells the applications whether the write was successful.

When several tags are to be written, one after another, each tag should be assigned a distinct EPC value. The Edge Server provides a mechanism for ensuring that each tag has a unique value. An EPC cache is a set of distinct EPC values, which may be used to provide EPC values to consecutive programming cycles without further application intervention. Applications receive cache reports (`EPCCacheReport`) to indicate when a cache is low or empty.

Programming Cycles

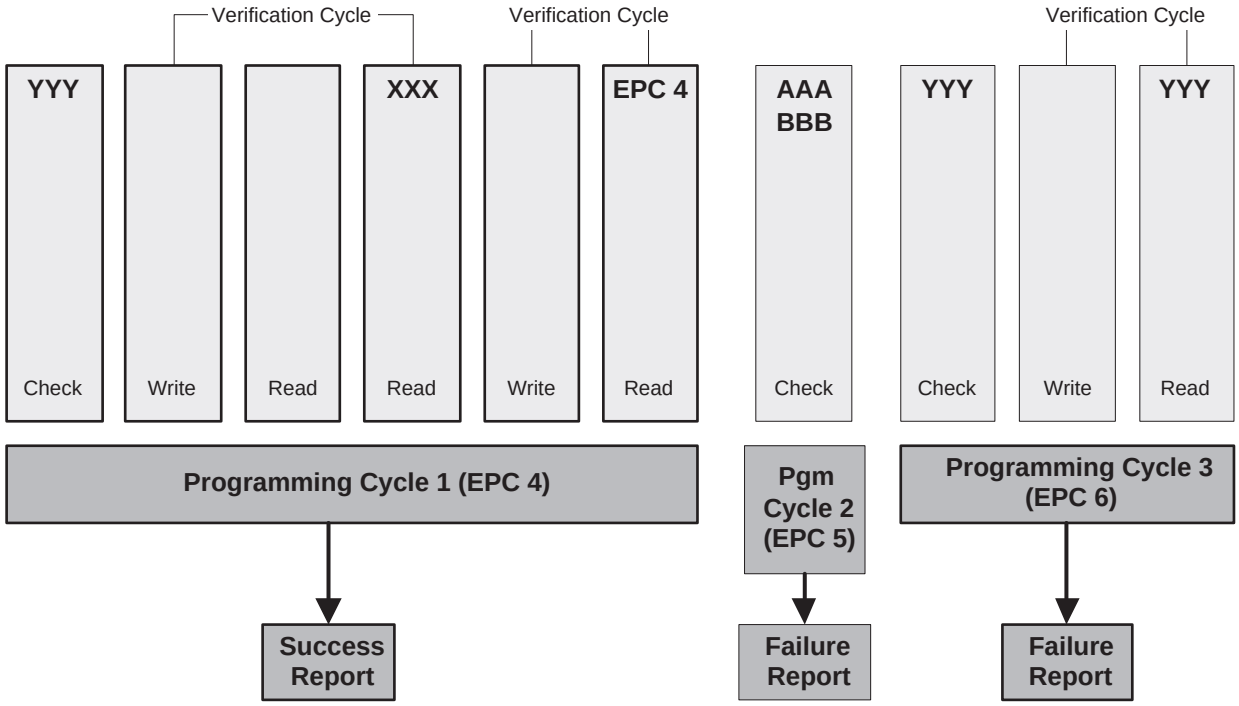
Tag writing services provided by the ALE API are organized around the notion of a programming cycle. Like an event cycle (see Read Cycles and Event Cycles on page 2-2), a programming cycle is an interval of time during which a specified operation takes place, at the conclusion of which a report is issued. And, like an event cycle, a programming cycle is specified through declarative specifications, in this case called programming cycle specifications (`PCSpec`).

A programming cycle is an interval of time during which a single tag is written and verified. Within a programming cycle, the reader attempts to ensure that there is a single tag in the field, write the tag, then read the tag to verify whether the write succeeded. The number of write attempts is configurable through parameters set within the `PCSpec`.

The overall pattern within a programming cycle consists of one or more "check" operations followed by one or more "verification cycles," each verification cycle consisting of a write attempt followed by one or more read attempts.

A check operation is a read carried out to verify that exactly one tag is in the field.

A verification cycle is said to succeed if the correct value is read from the tag, otherwise it fails. The programming cycle as a whole terminates successfully as soon as a successful verification cycle is completed.



In the diagram above, Programming Cycle 1 first checks that a single tag is in the field. It then performs a verification cycle, in which a write operation is performed followed by a read operation. In the example, the read operation shows no tags, so it is repeated, and a tag value other than what was written is seen. Thus, this verification cycle is deemed a failure. A second verification cycle is performed, which succeeds. Note that a "check" operation is not needed before the second verification cycle, because the first verification cycle's read operations serve to verify that a single tag is (still) in the field.

Programming Cycle 2's check operation sees two tags in the field, so the programming cycle immediately fails.

In Programming Cycle 3, each verification cycle's read shows the tag having a value other than the value written, so after `trials` attempts (see below), the programming cycle fails.

The following parameters govern execution of a programming cycle:

- `trials`

  The number of times an attempt is made to write the tag. If the `PCSpec` involves multiple logical readers, then each trial includes all logical readers.

- `duration`

  The total amount of time allotted to attempting to program the tag.

A programming cycle as a whole terminates when the first successful verification cycle completes, or after `trials` is exhausted, or `duration` elapses, or a stop trigger is received, whichever comes first.

Reader Implementation of Programming Cycle

Various reader manufacturers expose varying capabilities for tag writing. RFTagAware maps between the definition of programming cycles (described in Programming Cycles on page 2-6) and the actual capabilities available on various types of readers.

For example, one vendor's reader provides a "verify" function that can be used to probe the field for tags, even unprogrammed or invalid tags, but does not return distinct codes for "no tag" vs. "multiple tags in field". So this vendor's reader cannot directly implement RFTagAware's "check" notion. However, the vendor's "program" function performs RFTagAware's "check", "write" and "read" operations together, so RFTagAware can map its programming cycle onto the vendor's reader capabilities.

In general, RFTagAware presents a tag writing interface based on programming cycles as defined above, and maps programming cycles onto each kind of supported reader. Sometimes the mapping is exact, and other times the mapping is approximate but yields correct results.

EPC Caches and Pools

Programming cycles are designed to support a variety of use cases for tag writing. In the simplest case, an application makes an immediate request to write a single tag with a specified EPC value. In more complex cases, it is desirable for the ALE engine to write many tags (through consecutive programming cycles) without intervention by an application to specify the EPC value for each programming cycle. These use cases are handled through the use of EPC caches.

A `PCSpec` may be associated with an EPC cache, which is a collection of EPC values. Multiple `PCSpec` instances may share the same EPC cache. RFTagAware maintains the defined `PCSpec` instances and their EPC caches as part of its persistent state. Each time a `PCSpec` is activated, it takes the next EPC value from its EPC cache, and attempts to write that to a tag. When multiple `PCSpec` instances share a single cache, each will get a different EPC value each time it is activated. Hence, caches serve to ensure uniqueness of the EPC values written to tags.

When a `PCSpec`'s EPC cache has at least one EPC value available for writing, the cache is said to be *replenished*. When it has no EPC values, it is said to be *depleted*. A `PCSpec` whose EPC cache is depleted cannot program tags. Applications can receive asynchronous notifications when cache instances are depleted or nearing depletion. Applications may also add more EPC values to an existing cache (whether or not the cache is currently depleted) through the `replenishEPCCache` API operation. There is also an API operation called `depleteEPCCache` that removes all remaining IDs from a the EPC cache; this is useful when an application knows that it will no longer use the cache, and wishes to reclaim any unassigned EPC values for later use.

An application creates an EPC cache using the `defineEPCCache` operation, giving the name of the cache, and an `EPCCacheSpec` that specifies reporting parameters (these are described later). Optionally, initial contents of the EPC cache may be supplied to the `defineEPCCache` operation, in which case the EPC cache is initialized in the replenished state. Alternatively, an application may omit

the initial contents in the define operation, in which case it must later call `replenishEPCCache` in order for the `PCSpec` to be able to write tags.

Abstractly, an EPC cache is an ordered list of EPC values. Concretely, the ALE API provides a simple way to specify EPC caches using the EPC Pattern URN notation. An EPC cache is specified by an ordered list of Pattern URNs. Each Pattern URN represents a range of EPC values ordered lexicographically; the contents of the EPC cache is the concatenation of the ranges corresponding to Pattern URNs in the list.

Here are some examples of how to create tag caches.

This first example (GID-64-i tag format) is not in the EPC Tag Specification but will work with our simulator and help you to learn how to create tag patterns.

Pattern URNs	Cache Contents
urn:epc:pat:gid-64-i:1000.1000.1000 urn:epc:pat:gid-64-i:1000.1000.[2000-2002] urn:epc:pat:gid-64-i:1000.[100-101].[300-301]	urn:epc:tag:gid-64-i:1000.1000.1000 urn:epc:tag:gid-64-i:1000.1000.2000 urn:epc:tag:gid-64-i:1000.1000.2001 urn:epc:tag:gid-64-i:1000.1000.2002 urn:epc:tag:gid-64-i:1000.100.300 urn:epc:tag:gid-64-i:1000.100.301 urn:epc:tag:gid-64-i:1000.101.300 urn:epc:tag:gid-64-i:1000.101.301

Here is an example of creating a tag cache of SGTIN-64 tags.

Pattern URNs	Cache Contents
urn:epc:pat:sgtin-64:0.047400.126279.1 urn:epc:pat:sgtin-64:0.047400.126279.[10-13]	urn:epc:tag:sgtin-64:0.047400.126279.1 urn:epc:tag:sgtin-64:0.047400.126279.10 urn:epc:tag:sgtin-64:0.047400.126279.11 urn:epc:tag:sgtin-64:0.047400.126279.12 urn:epc:tag:sgtin-64:0.047400.126279.13

Here is an example of creating a tag cache of GID-96 tags.

Pattern URNs	Cache Contents
urn:epc:pat:gid-96:1000.1000.1000 urn:epc:pat:gid-96:1000.1000.[2000-2002] urn:epc:pat:gid-96:1000.[100-101].[300-301]	urn:epc:tag:gid-96:1000.1000.1000 urn:epc:tag:gid-96:1000.1000.2000 urn:epc:tag:gid-96:1000.1000.2001 urn:epc:tag:gid-96:1000.1000.2002 urn:epc:tag:gid-96:1000.100.300 urn:epc:tag:gid-96:1000.100.301 urn:epc:tag:gid-96:1000.101.300 urn:epc:tag:gid-96:1000.101.301

Typically, only one component of the pattern is a range.

Note that while the EPC values generated from any one EPC Pattern URN are distinct and in ascending order, different patterns used to replenish the same cache may overlap or appear in non-ascending sequence. If an application wishes to ensure uniqueness of EPCs generated from the same

cache (as is commonly the case), the application must always replenish the cache with unique patterns.

Reports

When a programming cycle completes, it sends a report to interested applications to say what happened. Unlike event cycle reports, however, the reports issued by a programming cycle are more limited in nature.

The RFTagAware Edge Server's tag programming facility can issue two kinds of report:

- Write report

  Issued when a programming cycle completes, either successfully having written a tag value or because an error occurred. For a successful tag write, the report includes the EPC value that was written. For a failed tag write, the report contains information describing the error.

- Cache report

  Issued when an EPC cache's number of remaining EPC values drops to (or below) a specified level.

`PCSpec` instances do not contain any parameters describing the write reports to be generated; this is in contrast to `ECSpec` instances which include one or more `ECReportSpec` instances describing the various reports generated by event cycles. `EPCCacheSpec` instances do contain parameters that describe the cache reports to be generated, including the threshold at which a cache report should be generated.

Comparison of Event Cycles and Programming Cycles

RFTagAware's approaches to tag reading (as embodied in event cycles) and tag writing (as embodied in programming cycles) are very similar, but differ in certain respects. The following table summarizes the similarities and differences of event cycles and programming cycles.

	Event Cycle	Programming Cycle
Direction	The flow of tag data is in one direction: from tag, through the RFTagAware Edge Server, to application.	The flow of tag data is bidirectional: tag data flows from application to the Edge Server to say what tag ID should be written; tag data flows back from a tag through the Edge Server to the application when the write to the tag is verified.
Readers	One or more logical readers; each may be a single antenna or multiple antennas or a composite of other logical readers	Same, but multiple antennas are used differently. In the reading case, the event cycle combines the set of tags seen by all of the antennas. In the writing case, the programming cycle tries writing with each antenna in turn until the tag is successfully written. The "check" and "read" operations (see above) are carried out using all antennas.
Cycle start condition	Start trigger, repeat interval, or immediate/poll by application	Start trigger, or immediate/poll by application. There can be no repeat interval.
Cycle end condition	Stop trigger, duration, stable field interval, or all applications unsubscribe	Successful tag write, unless stopped first by stop trigger, duration, trials, application undefine/ suspend.
Reports	One or more tag read reports, each specifying report type, report set, report groups, and filters	Reports of specific events, including: • Successful tag write • Failed tag write • EPC cache level low
Report Subscriptions	Applications that want to know what tags are in the field.	Two kinds of subscription: • Applications that subscribe to write reports are notified whenever a tag programming operation completes. • Applications that subscribe to low-cache reports are notified when the cache is low.
API operations	Define, undefine, subscribe, unsubscribe, poll, immediate	PCSpec operations: define, undefine, subscribe, unsubscribe, poll, immediate. EPCCacheSpec operations: defineEPCCache, undefineEPCCache, replenishEPCCache, depleteEPCCache, subscribeEPCCache, unsubscribeEPCCache.

Specifying Readers to the Edge Server

Configuring Readers

You configure the Edge Server to recognize your physical readers by editing the `edge.props` file or entering configuration information via the Admin Console. The physical reader driver parameters, acceptable values, and defaults for readers recognized by RFTagAware are covered in detail in the *RFTagAware Reader Configuration Guide.*

Physical Readers vs. Logical Readers

In specifying an event cycle or programming cycle, an application names one or more readers of interest. This is necessary because a single Edge Server may manage many readers that are used for unrelated purposes. For example, in a large warehouse, there may be three readers covering each of ten loading dock doors; in such a case, a typical ALE request may be directed at the three readers for a particular door, but it is unlikely that an application tracking the flow of goods into trucks would want the reads from all 30 readers to be combined into a single event cycle.

This raises the question of how applications specify which readers are to be used for a given cycle. One possibility is to use identities associated with the readers themselves; e.g., a unique name, serial number, IP address, etc. This is undesirable for several reasons:

- The exact identities of readers deployed in the field are likely to be unknown at the time an application is authored and configured.
- If a reader is replaced, this unique reader identity will change, forcing the application configuration to be changed.
- If the number of readers must change — for example, because it is discovered that four readers are required instead of three to obtain adequate coverage of a particular loading dock door — then the application must be changed.

To avoid these problems, ALE introduces the notion of a "logical reader." Logical readers are abstract names that an application uses to refer to one or more readers that have a single logical purpose; e.g., readers positioned around a door might be called DockDoor42. Logical readers may be usefully thought of as being equivalent to "locations." Within the Edge Server, an association is maintained between logical names such as DockDoor42 and the physical readers assigned to fulfill that purpose. Any event cycle or programming cycle specification that refers to DockDoor42 is understood by the Edge Server to refer to the physical reader (or readers) associated with that name.

In many cases, a single RFID reader may support the use of more than one antenna, with the ability to treat each antenna independently. The Edge Server permits such readers to be configured so that each antenna is exposed through ALE as a separate logical reader. This gives flexibility to applications to use antennas independently or in concert, by simply specifying one or more logical readers in an `ECSpec`.

Adding a Transient Filter

You can apply a transient (tag) filter to any logical reader that you configure in `edge.props`. Different logical readers may share the same filter settings, or have different settings.

For each transient filter you add, three parameters that control its operation:

- `minReads` – The number of times a tag must be read before being included in the filter (i.e., visible to the event cycle).
- `firmInterval` – The maximum time (in milliseconds) allowed between reads that increase the `minReads` count.
- `expiredInterval` – The maximum duration (in milliseconds) for a tag not to be read before expiring from the filter.

These parameters control the filter in the following way. A tag is considered "soft" until it has been read `minReads` times, with no more than `firmInterval` milliseconds passing between each of those reads. A "soft" tag is not included in the filter's output, and therefore will not be considered by any active event cycles. If a "soft" tag is not read for more than `firmInterval` milliseconds, then the count starts over again the next time the tag is read.

When the count reaches `minReads`, the tag becomes "firm". A "firm" tag is included in the filter's output, and will be considered by any active event cycles that use this logical reader. A "firm" tag remains "firm" even if the tag is not read in every read cycle, until it is not read for `expiredInterval` milliseconds. When that happens the tag is considered "expired." The next time the tag is read, it will be considered "soft", and the filter process starts again.

When all event cycles associated with a reader enter the unrequested state, then all filters for that reader will be reset. This means that all counts will be reset to zero, and all tags will be considered "soft" the next time they are seen. Setting `firmInterval` to -1 means that the count for a given tag will continue to increase towards `minReads` regardless of the time between reads, until the filter is reset. Setting `expiredInterval` to -1 causes any "firm" tag will remain "firm" until the filter is reset.

When choosing values for `firmInterval` and `expiredInterval`, you must be aware of the rate at which the logical reader performs read cycles. For most physical reader types, this is the `defaultRate` parameter times the number of active logical readers. If the `firmInterval` is less than this, then tags will never become "firm" and no tags will be reported to any event cycle. Likewise, if the `expiredInterval` is less than the `defaultRate` parameter times the number of active logical readers, then it is equivalent to specifying an `expiredInterval` of zero.

To add a transient filter to a logical reader, add the following settings in the `edge.props` file. First, add the following lines to define a named filter (in the example, the filter is named `myfilter1`):

```
com.connecterra.ale.filter.myfilter1.class=
   com.connecterra.ale.filtertypes.TransientFilterFactory
com.connecterra.ale.filter.myfilter1.minReads = 3
com.connecterra.ale.filter.myfilter1.firmInterval = 1400
com.connecterra.ale.filter.myfilter1.expiredInterval = 1400
```

Then, for each logical reader to which you want to add the filter, add a line like this (in the example, the logical reader is named `myreader`):

```
com.connecterra.ale.logicalReader.myreader.filters = myfilter1
```

To apply the same filter parameters to more than one logical reader, you may specify the same filter name for more than one reader. Even though more than one logical reader refers to the same filter name, each logical reader is processed by a different filter instance.

Using Composite Readers

You can create additional logical readers by combining existing logical readers. A logical reader created in this way is called a *composite reader*. By defining composite readers, you can decouple applications from decisions you take at deployment time about how many readers are needed to cover a single location.

For example, suppose that you have four logical readers covering a location called LoadingDock23:

- `LoadingDock23_Reader1`
- `LoadingDock23_Reader2`
- `LoadingDock23_Reader3`
- `LoadingDock23_Reader4`

You specify these reader names in each `ECSpec` that you create for LoadingDock23.

Then suppose you discover that you really need five readers to cover that location. If you specified single logical reader names in each `ECSpec`, then you would need to:

- Reconfigure the RFTagAware Edge Server to add the fifth reader.
- Go back and edit each `ECSpec` to include the new fifth reader.

Changing every `ECSpec` is undesirable, especially if some applications generate `ECSpec` instances for LoadingDock23 on the fly.

The alternative is to define a composite reader called `LoadingDock23`. Initially, this composite reader is configured to contain `LoadingDock23_Reader1` through `LoadingDock23_Reader4`. Applications that want to get data from all readers in LoadingDock23 simply specify `LoadingDock23` as the sole logical reader in their `ECSpec` instances.

Then, when you add your fifth reader, all you have to do is:

- Edit the `edge.props` file or use the Administration Console RFID Devices node to:
  - Add the fifth reader.
  - Change the definition of the `LoadingDock23` composite reader to include the fifth reader.
- Leave your `ECSpec` instances unchanged.

Chapter 3: Asynchronous Notification Mechanisms

Contents

This chapter describes the asynchronous notification mechanisms RFTagAware uses to deliver reports.

Overview

Applications may define specifications (`ECSpec`, `PCSpec`, and `EPCCacheSpec`) and later subscribe for asynchronous delivery of corresponding reports (`ECReports`, `PCWriteReport`, `EPCCacheReport`). RFTagAware provides a number of ways to deliver asynchronous reports. When an application makes a new subscription, it specifies a delivery address in the form of a Uniform Resource Identifier (URI). The URI specifies a particular method of notification delivery, and provides parameters that further identify the receiver.

The RFTagAware Edge Server provides an extensible mechanism for adding new delivery mechanisms. RFTagAware also provides five out-of-box event delivery drivers that are applicable in a wide range of circumstances. The following sections define the delivery mechanisms supported out-of-box by RFTagAware, and describe how to construct the URIs to provide to the subscribe method of the ALE interface in order to use them.

All of the notification delivery mechanisms described below encode reports into XML. For additional information, see:

- XML Representations on page 5-25 (for `ECSpec` and `ECReports` objects)
- XML Representations on page 6-12 (for `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` objects)

XML via HTTP POST

The Edge Server may deliver reports by sending an HTTP POST request, where the payload is the report instance encoded using XML. The general form of the subscription URI is:

```
http://host:port/remainder-of-URL
```

`host`	`host` is the DNS name or IP address of the host where the receiver is listening for incoming HTTP connections.
`port`	`port` is the TCP port on which the receiver is listening for incoming HTTP connections. The port and the preceding colon character may be omitted, in which case the port defaults to 80.
`remainder-of-URL`	`remainder-of-URL` is the URL path to which the HTTP POST operation will be directed.

The payload of the notification is the report instance (`ECReports`, `PCWriteReport`, `EPCCacheReport`) for the subscribed event or programming cycle, encoded into XML.

The response code returned by the HTTP server is used to determine whether the notification succeeded or not. A response code of 200 through 299 is considered successful; any other response code is considered a failure.

XML via TCP Socket

The Edge Server may deliver reports by opening a TCP socket to a designated receiver, sending an XML report, then closing the connection. The general form of the subscription URI is:

```
tcp://host:port
```

`host`	`host` is the DNS name or IP address of the host where the receiver is listening for incoming TCP socket connections.
`port`	`port` is the TCP port on which the receiver is listening for incoming TCP socket connections.

The payload of the notification is the report instance (`ECReports`, `PCWriteReport`, `EPCCacheReport`) for the subscribed event or programming cycle, encoded into XML.

XML via JMS Message

The Edge Server can deliver event cycle reports by sending a JMS Message to a JMS Topic or a JMS Queue where the message is a `javax.jms.TextMessage` that contains the `ECReports` instance encoded using XML.

The general form of the URI is:

```
jms:/topic/conn_factory/topic_name>[?queryParams]
jms:/queue/conn_factory/queue_name>[?queryParams]
```

Optional segments are enclosed in square brackets [].

URI Segment	Optional/ Required	Description
`topic | queue`	Required	Indicates whether the JMS notification driver adds messages to a queue or if it publishes messages to a topic.
`conn_factory`	Required	The JNDI name of the connection factory for obtaining a topic/queue connection.

URI Segment	Optional/ Required	Description
`topic_name|queue_name`	Required	The name of the topic or queue to which the JMS notification driver sends its messages.

`queryParams`	Optional	You can specify additional URI query parameters if needed. To specify just one query parameter, append a string like `?param1=value1` to the URI string. When you need to specify more than one parameter, append a string like `?param1=value1&param2=value2&param3=value3`
		You can use the following query parameters:
		• `username`
		The user name that is used when creating a JMS topic connection or queue connection. Also see the password query parameter, below.
		• `password`
		The password of the user specified in the URI.
		• `ackMode`
		The acknowledgement mode that is used when the queue or topic session is created. Recognized values are:
		`auto`
		`client`
		`dups_ok`
		If you do not specify `ackMode`, then a default value of `auto` is used.
		• Query parameters whose names start with `jndi:` are added to the `javax.naming.Context` environment when one is constructed to access a naming service to perform the necessary JNDI lookups. If `javax.naming.Context` properties are specified in the URI as well as being configured on the Edge Server, then those properties that are specified in the URI will override the ones configured on the Edge Server. (See Setting up the JMS Notification Driver on page 3-6).
		• If you specify any other parameters, they will be added to the `javax.jms.TextMessage` as `String` properties, where the query parameter name is the property name and the query parameter value is the property value

The Edge Server also adds three additional properties to the `TextMessage`:

`specName`	The name of the `ECSpec` associated with this `ECReport`.
`savantID`	The Edge Server ID that is originating this `ECReport`.
`date`	The date of the `ECReport`. A time in milliseconds, as returned from `System.currentTimeMillis()`.

The driver sends JMS messages in a non-transactional context.

Note: All string values in the various segments of the URI have to be properly URI escaped. For example, to specify a forward slash (/) character in the URI, where that character is part of either the JNDI name or the connection factory, or the JNDI name of the

queue or the topic, you need to use %2F instead of the forward slash character. See RFC 2396 for more information (`http://www.ietf.org/rfc/rfc2396.txt`).

Examples

The following example causes the Edge Server to send XML reports via JMS to a queue named `MyECReportSpecQueue`. The JNDI/Naming service is accessed using `iiop` at `jms.example.com` via port 1099. The JNDI name of the connection factory to be used is `ConnectionFactory`.

```
jms:/queue/ConnectionFactory/MyECReportSpecQueue?jndi:java.naming.provider.url=
iiop://jms.example.com:1099
```

The next example causes the Edge Server to send XML reports via JMS to a topic named `MyECReportSpecTopic`. The JNDI/Naming service is accessed using `iiop` at `jms.example.com` via port 1099. The JNDI name of the connection factory to be used is called `ConnectionFactory`.

```
jms:/topic/ConnectionFactory/MyECReportSpecTopic?jndi:java.naming.provider.url=
iiop://jms.example.com:1099
```

Here is a more complex example. This example incorporates JMS security as well as JNDI/Naming service security while using a queue. It uses the `jndi:` query parameters to set a security principal and security credential of `guest/PasswordForGuest` to access the JNDI/Naming service, and uses the username/password combination of `bob/PasswordForBob` to open a connection to the queue.

```
jms:/queue/ConnectionFactory/MyECReportSpecQueue?username=bob&password=PasswordForBob&
jndi:java.naming.provider.url=iiop://jms.example.com:1099&
jndi:java.naming.security.principal=guest&
jndi:java.naming.security.credentials=PasswordForGuest
```

The next example uses all the elements of the previous example while overriding the default naming context factory class being used by the JMS driver. The class `org.jnp.interfaces.NamingContextFactory` will be used instead of the one the driver is configured with when performing JNDI lookups.

```
jms:/queue/ConnectionFactory/
MyECReportSpecQueue?username=bob&password=PasswordForBob&secPrincipal=guest&jndi:java.
naming.provider.url=iiop://
jms.example.com:1099&jndi:java.naming.security.principal=guest&jndi:java.naming.securi
ty.credentials=PasswordForGuest&jndi:java.naming.factory.initial=org.jnp.interfaces.Na
mingContextFactory
```

The next example adds additional query parameters, `field1` and `field2` that will be added onto the JMS text message.

```
jms:/queue/ConnectionFactory/
MyECReportSpecQueue?username=bob&password=PasswordForBob&secPrincipal=guest&jndi:java.
naming.provider.url=iiop://
jms.example.com:1099&jndi:java.naming.security.principal=guest&jndi:java.naming.securi
ty.credentials=PasswordForGuest&jndi:java.naming.factory.initial=org.jnp.interfaces.Na
mingContextFactory&field1=value1&field2=value2
```

Setting up the JMS Notification Driver

To use the JMS Notification driver, you need to edit the following two Edge Server configuration files. Both are in the `etc` directory:

- `edge.props`
- `jms.options`

For information on how to edit these files, see the *RFTagAware Deployment Guide*.

XML Written to a File

The Edge Server may deliver event or programming cycle reports by creating or appending XML to files in the Edge Server's local file system. The general form of the subscription URI is:

```
file:///filename
```

where `filename` is either the name of a file, a directory, or a pattern as described below.

If `filename` names a specific file that already exists, the `ECReports`, `PCWriteReport`, or `EPCCacheReport` instance is encoded as XML and appended to the file. If more than one cycle completes, the resulting file is not a well-formed XML document, but a concatenation of XML documents.

If `filename` does not name an existing file but the directory portion names an existing directory, then the file is created, and the `ECReports`, `PCWriteReport`, or `EPCCacheReport` instance is encoded as XML and written to the file. If another cycle completes, the prior case applies and the file will be a concatenation of XML documents.

If `filename` names a directory, then the `ECReports`, `PCWriteReport`, or `EPCCacheReport` instance is written as a new file in that directory with a unique name of the form:

```
specName-yyyyMMddhhmmssSSS.xml
```

where `specName` is the name of the `ECSpec`, `PCSpec`, or `EPCCacheSpec` that defined the cycle, and `yyyyMMddhhmmssSSS` is the timestamp in the `ECReports`, `PCWriteReport`, or `EPCCacheReport` instance, in the local timezone (SSS is the millisecond, which helps insure the uniqueness of the filename even if several reports are generated per second).

If `filename` contains parentheses, the text within the parentheses is considered to be a pattern string for the timestamp, and the resulting filename after substitution is treated as above. For example, given this subscription URI:

```
file:///mydir/myprefix-(yyyy-MM-dd).xml
```

then all reports generated on Christmas Day 2003 would be appended to the file

```
/mydir/myprefix-2003-12-25.xml
```

In all cases, the XML is as described in XML Representations on page 5-25 and XML Representations on page 6-12.

XML Displayed on the Edge Server Console

The Edge Server may deliver event cycle reports by displaying XML in the console window where the Edge Server was started. This is typically useful only in debugging situations. The general form of the subscription URI is:

```
console:heading
```

where `heading` is an arbitrary text string (conforming to URI syntax restrictions). The heading is printed prior to each report instance (`ECReports`, `PCWriteReport`, or `EPCCacheReport`). This may be useful to distinguish reports arising from different subscriptions. URI syntax restrictions prohibit the heading from being empty.

The Null Delivery Method

The Edge Server permits a null subscription URI, having the following form:

```
null:ignoredText
```

where `ignoredText` is any non-empty text string (it is ignored by the Edge Server, but is necessary to conform to URI syntax restrictions). Subscribing an `ECSpec` to this URI will cause the `ECSpec` to activate, but the reports will not be delivered anywhere (unless there are other subscribers to the same `ECSpec`). This URI is used when you want to force the readers and the Edge Server to perform activity, but you are not interested in the results. Typically this need arises only in testing situations.

Chapter 4: Triggers

Contents

Introduction

Applications may define event cycle specifications (`ECSpec`) and programming cycle specifications (`PCSpec`) where the beginning and/or end of each cycle is triggered by external events. RFTagAware provides an extensible mechanism for connecting sources of external events to the ALE engine.

OLE for Process Control (OPC) Trigger Driver

OPC is a series of standards specifications that define a standard set of objects, interfaces and methods for use in process control and manufacturing automation applications to facilitate interoperability. (For more information on OPC, see `http://www.opcfoundation.org`.)

RFTagAware includes a driver for OPC triggers. An event or programming cycle in the Edge Server may be triggered by polling for a change in an OPC item. The Edge Server communicates with the OPC service using the OPC XML-DA protocol, which is a SOAP interface to an OPC Data Access provider. The OPC XML-DA implementation is provided by a third party.

The general form of the trigger URI is:

```
opcpoll:itemName=item;http://hostname/location
```

where:

- `item` is the name of the OPC item that the driver polls for changes.
- `http://hostname/location` is the URL that is used to create the connection to the OPC OPC XML-DA server.

When an event cycle or programming cycle that uses an OPC trigger is first requested (that is, when the event cycle or programming cycle is invoked using the `poll` or `immediate` method or subscribed using `subscribe`), the Edge Server polls the OPC XML-DA server for the current value of the specified OPC item. Afterwards, each time the value of the OPC item changes, a trigger will be delivered to the event cycle or programming cycle.

Additional Trigger Drivers

Please contact ConnecTerra support (see Contacting Technical Support on page viii) for more information about obtaining additional trigger drivers.

Chapter 5: Reading Tags Using the ALE API

Contents

This chapter describes the ALE API programming components you use to read tags.

Introduction to the ALE API Specification

This section provides a formal, abstract specification of the ALE API for reading tags. The external interface is defined by the `ALE` class (See ALE: Main Tag Reading Interface on page 5-3). This interface makes use of a number of complex data types that are documented in the sections starting at ECSpec on page 5-6. The ALE API is compliant with the EPCglobal ALE 1.0 specification.

The general interaction model is that there are one or more clients that make method calls to the ALE interface. Each method call is a request, which causes the ALE engine to take some action and return results. Thus, methods of the ALE interface are synchronous.

The ALE interface also provides a way for clients to subscribe to events that are delivered asynchronously. This is done through methods that take a URI as an argument. Such methods return immediately, but subsequently the ALE engine within the Edge Server may asynchronously deliver information to the consumer denoted by the URI argument.

In the sections below, the API is described using UML class diagram notation, as shown below:

```
dataMember1 : Type1
dataMember2 : Type2
---
method1(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
method2(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
```

The box as a whole refers to a conceptual class, having the specified data members and methods. Within the UML descriptions, data members/methods are marked as belonging to one of the following categories:

- The EPCglobal ALE specification.
- RFTagAware extensions to the EPCglobal ALE specification.

The ALE API is realized in several equivalent forms within RFTagAware:

- There is a binding of the ALE API to a SOAP web service, described by a WSDL file.
- The complex data types have a standard representation as XML documents, defined by an XSD schema.
- There is a binding of the ALE API to Java, in which it takes the form of a collection of Java interface and class definitions.

Each of these concrete forms of the ALE API has a slightly different structure and gives slightly different names to the different conceptual classes, data members, and methods defined in UML within this section. This is unavoidable, owing to syntactic constraints and stylistic norms within these different implementation technologies.

In most cases, the mapping from conceptual UML to the concrete details of any particular binding is very straightforward; where it is not, the specific documentation for each binding will make clear the relationship to the UML. The UML-level descriptions in this section should be considered normative.

- For specifics of the Java binding, see the Javadoc that is included in the RFTagAware installation.
- For specifics of the WSDL binding, see the WSDL file:
  `EPCglobal-ale-1_0.wsdl`

  This file is located in your RFTagAware installation directory under `share/schemas`.
- For specifics of the XML representation of the complex data types, see the following XSD files:
  - `EPCglobal-ale-1_0.xsd`

    Defines EPCglobal ALE schema; references RFTagAware extensions.
  - `EPCglobal.xsd`

    Defines the EPCglobal common types, `Document` and `EPC`, referred to by `EPCglobal-ale-1_0.xsd.`
  - `EPCglobal-ale-1_0-RFTagAware-extensions.xsd`

    Defines the RFTagAware schema extensions.

  These files are located in your RFTagAware installation directory under `share/schemas`.

  See also XML Representations on page 5-25.

ALE: Main Tag Reading Interface

Java implementation package: `com.connecterra.ale.api`

EPCglobal ALE

```
define(ecSpecName:string, spec:ECSpec) : void
undefine(ecSpecName:string) : void
getECSpec(ecSpecName:string) : ECSpec
getECSpecNames() : List   // returns a List of strings naming ECSpec instances
subscribe(ecSpecName:string, notificationURI:URI) : void
unsubscribe(ecSpecName:string, notificationUri:URI) : void
getSubscribers(ecSpecName:string) : List   // returns a List of subscriber URIs
poll(ecSpecName:string) : ECReports
immediate(spec:ECSpec) : ECReports
getStandardVersion() : string
getVendorVersion() : string
```

RFTagAware Extensions

```
getECSpecInfo(ecSpecName:string) : ECSpecInfo
redefine (ecSpecName:string, spec:ECSpec) : void
subscribe(ecSpecName: string, notificationURI: URI, controls:
ECSubscriptionControls) : void
suspend (ecSpecName:string) : void
unsuspend (ecSpecName:string) : void
listLogicalReaderNames() : List
// returns a List of Strings in sorted order naming all logical readers known to the ALE engine
getECSubscriptionInfo(ecSpecName:string, notificationURI:URI):ECSubscriptionInfo
```

An `ECSpec` is a complex type that defines how an event cycle is to be calculated.

There are two ways to cause event cycles to occur:

- A standing `ECSpec` may be posted using the `define` method. Subsequently, one or more clients may subscribe to that `ECSpec` using the `subscribe` method. The `ECSpec` will generate event cycles as long as there is at least one subscriber.

  A `poll` call is like subscribing then unsubscribing immediately after one event cycle is generated (except that the results are returned from `poll` instead of being sent to a URI).

- An `ECSpec` can be submitted for immediate execution using the `immediate` method. This is equivalent to defining an `ECSpec`, performing a single `poll` operation, and then undefining it.

The execution of `ECSpec` instances is defined formally as follows. `ECSpec` instances are each in one of three states: `unrequested`, `requested`, and `active`. An `ECSpec` is said to be `requested` if any of the following is true:

- It has previously been defined using `define`, it has not yet been `undefined`, and there has been at least one `subscribe` call for which there has not yet been a corresponding `unsubscribe` call.
- It has previously been defined using `define`, it has not yet been undefined, a `poll` call has been made, and the first event cycle since the `poll` was received has not yet been completed.
- It was defined using the `immediate` method, and the first event cycle has not yet been completed.

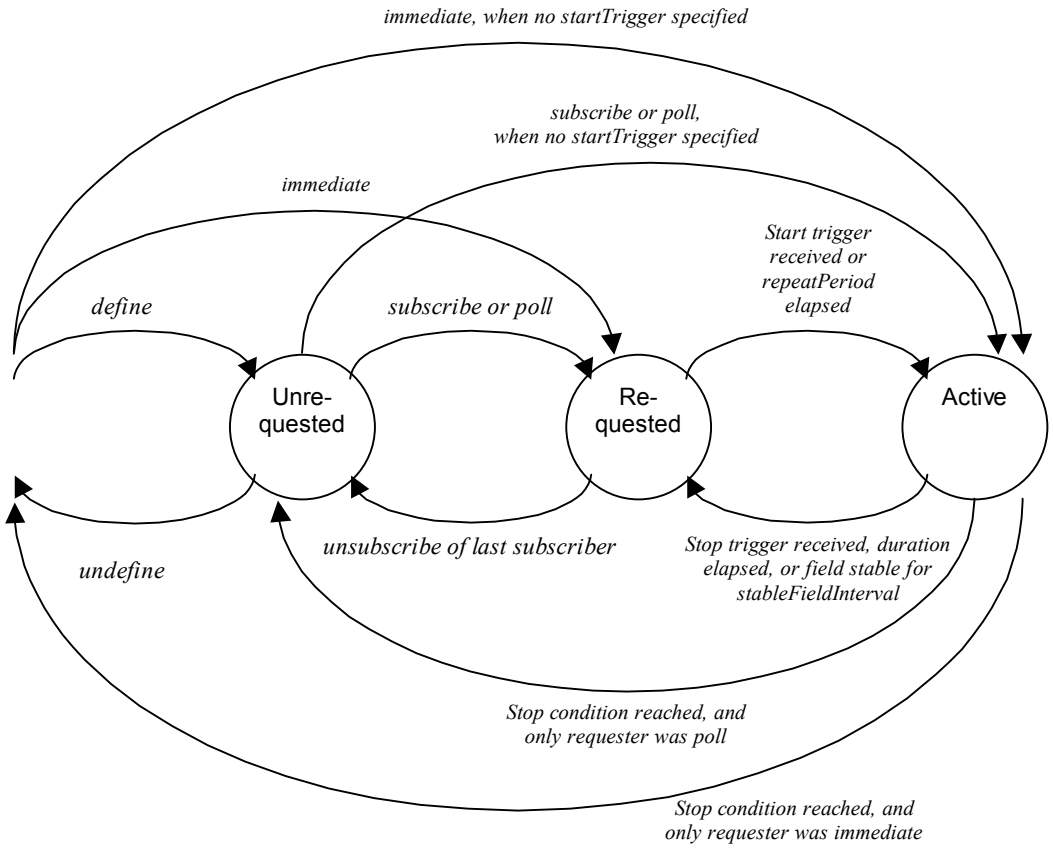
Once requested, an `ECSpec` is said to be `active` if reads are currently being accumulated into an event cycle based on the `ECSpec`. Standing `ECSpec` instances that are requested using `subscribe` may transition between `active` and `inactive` multiple times. `ECSpec` instances that are requested using `poll` or created using `immediate` will transition between `active` and `inactive` just once (though in the case of `poll`, the `ECSpec` remains defined afterward so that it could be subsequently polled again or subscribed to).

Two other methods are provided to manipulate `ECSpec` instances while preserving existing subscriptions:

- The `suspend` and `unsuspend` methods let you temporarily "suspend" an `ECSpec` without removing its subscriptions. While an `ECSpec` is suspended, you can add and remove subscriptions using the `subscribe` and `unsubscribe` methods, but the `ECSpec` behaves as though it is in the unrequested state — it causes no read cycles to take place, and generates no `ECReports`.
- The `redefine` method lets you replace the definition of an `ECSpec`. It is roughly equivalent to unsubscribing all subscribers, undefining the `ECSpec`, defining a new `ECSpec` with the same name, then replacing the subscribers. The `redefine` method is intended for development and not production use, as it may cause a gap in event cycle processing.

State Diagram

The effects of these methods on an `ECSpec` is summarized in the state diagram below.



The two methods `getStandardVersion` and `getVendorVersion` provide information about compliance with EPCglobal specifications:

- `getStandardVersion` returns a string that identifies what version of the EPCglobal ALE specification this implementation complies with. In RFTagAware 1.3, this method always returns the string `1.0`.
- `getVendorVersion` returns a string that identifies what vendor extensions this implementation provides. In RFTagAware 1.3, this method always returns the string: `http://version.connecterra.com/ALE/1`

Primary ECSpec Data Types

The primary data types associated with the ALE API are

- `ECSpec`, which specifies how an event cycle is to be calculated and reported
- `ECReports`, which contains one or more reports generated from one activation of an `ECSpec`.

  `ECReports` instances are returned from the `poll` and `immediate` methods, and also sent to URIs when `ECSpec` instances are subscribed to using the `subscribe` method.

For detailed information on the `ECSpec` and `ECReports` data types, see ECSpec on page 5-6 and ECReports on page 5-17.

ECSpec

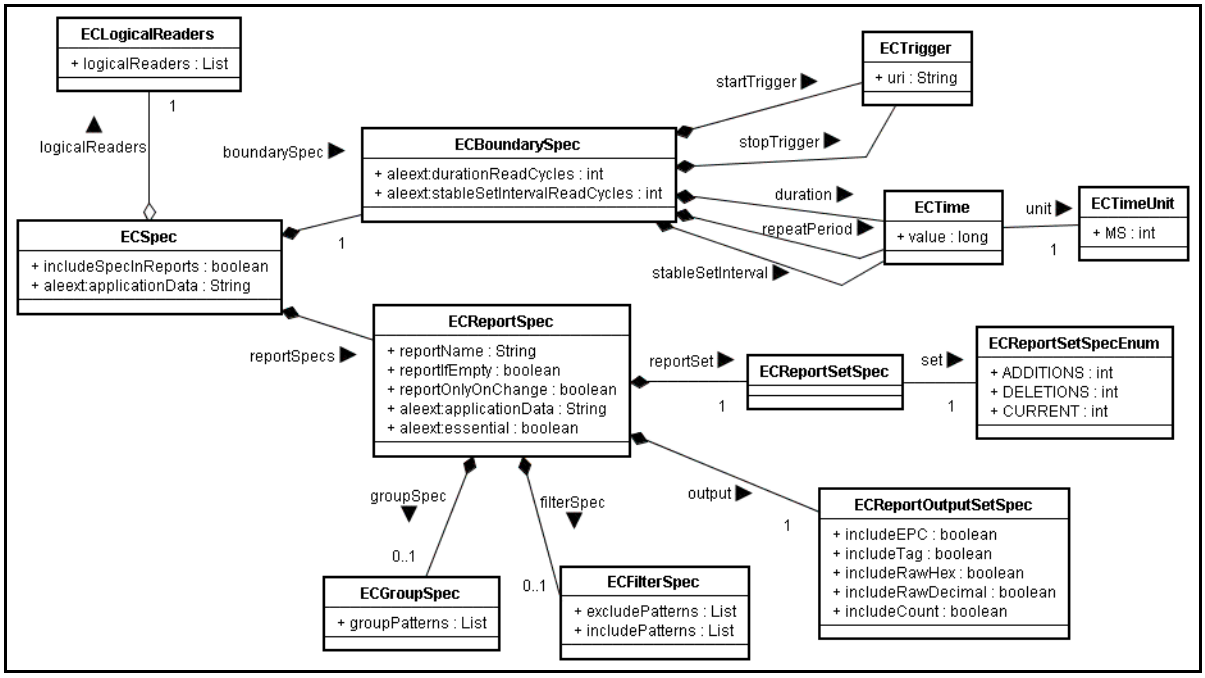
Java implementation package: `com.connecterra.ale.api`

An `ECSpec` is a complex type describing an event cycle and one or more reports that are to be generated from it. It contains:

- A list of readers whose reader cycles are to be included in the event cycle. Each member of this list may be a single logical reader, or the name of a composite reader. For information on composite readers, see Using Composite Readers on page 2-14.
- A specification of how the boundaries of event cycles are to be determined.
- A list of report specifications, each of which describes a report to be generated from this event cycle.
- A boolean value indicating whether or not to include the complete `ECSpec` as part of every `ECReports` instance generated by this `ECSpec`.

It also contains an optional "application data" string, which is simply copied unmodified into every `ECReports` instance generated from this `ECSpec`.

ECSpec UML Diagram



EPCglobal ALE

```
readers : List   // List of logical or composite reader names
boundaries : ECBoundarySpec
reportSpecs : List   // List of one or more ECReportSpec instances
includeSpecInReports : boolean
```

RFTagAware Extensions

```
applicationData : string
---
```

Java Implementation Notes: Note that in the Java API, `ECSpec` does not include a `boundaries` data member that references an `ECBoundarySpec`. Rather, in Java, `ECSpec` provides getter and setter methods for accessing `ECBoundarySpec`'s data members (`startTrigger`, `repeatPeriod`, etc.) directly. See the Javadoc and ECBoundarySpec on page 5-7.

ECBoundarySpec

An `ECBoundarySpec` specifies how the beginning and end of event cycles are to be determined.

EPCglobal ALE

```
startTrigger : ECTrigger
repeatPeriod : ECTime
stopTrigger : ECTrigger
duration : ECTime
```

```
stableSetInterval : ECTime
RFTagAware Extensions
durationReadCycles: int
stableSetIntervalReadCycles: int
---
```

The time values `duration` and `stableSetInterval` can be expressed in either of two units: milliseconds or read cycles. One read cycle unit denotes whatever interval of time is required to complete one read cycle for every reader that is included in the event cycle. This may be dynamic with the number of physical reading elements in the Logical Reader. The time values must be non-negative. Zero means "unspecified" (in which case the value of the corresponding units argument is irrelevant).

`startTrigger` and `repeatPeriod` are mutually exclusive.

The conditions under which an event cycle is started depends on the settings for `startTrigger` and `repeatPeriod`:

- If `startTrigger` is specified, an event cycle is started when:
  - The `ECSpec` is in the requested state and the specified start trigger is received.
- If `startTrigger` is not specified and `repeatPeriod` is specified, an event cycle is started when
  - The `ECSpec` transitions from the *unrequested* state to the *requested* state; or
  - The `repeatPeriod` has elapsed from the start of the last event cycle, and in that interval the `ECSpec` has never transitioned to the *unrequested* state.
- If neither `startTrigger` nor `repeatPeriod` are specified, an event cycle is started when:
  - The `ECSpec` transitions from the *unrequested* state to the *requested* state; or
  - Immediately after the previous event cycle, if the `ECSpec` is in the *requested* state.

An event cycle, once started, extends until one of the following is true:

- The `duration` or `durationReadCycles`, when specified, expires.
- When the `stableSetInterval` or `stableSetIntervalReadCycles` is specified, no new EPCs have been reported by any reader in the specified interval.
- The `stopTrigger`, when specified, is received.
- The `ECSpec` transitions to the *unrequested* state.

Note that the first of these conditions to become true terminates the event cycle. For example, if both `duration` and `stableSetInterval` are specified, then the event cycle terminates when the `duration` expires, even if the reader field has not been stable for `stableSetInterval`. But if the field is stable for `stableSetInterval`, the event cycle terminates even if the total time is shorter than the specified

duration. Likewise, if both `duration` and `durationReadCycles` are specified, the event cycle terminates when the first of these time periods elapses.

In all the descriptions above, note that an `ECSpec` presented via the `immediate` method means that the `ECSpec` transitions from *unrequested* to *requested* immediately upon calling `immediate`, and transitions from *requested* to *unrequested* immediately after completion of the event cycle.

URIs specify an event cycle's start or stop triggers. Please contact ConnecTerra customer support for more information about trigger URIs.

It is possible to specify both `duration` and `stableSetInterval` in units of milliseconds and/or units of read cycles. Be careful when using units of read cycles. If any reader experiences a failure during the event cycle, it will not complete read cycles, and hence time as measured in read cycles may never reach the limits set for `duration` or `stableSetInterval`. For this reason, it is highly recommended to include a `duration` in units of milliseconds, to act as an overall timeout for the event cycle.

ECBoundarySpec Implementation Notes

Java: `ECBoundarySpec`, `ECTime`, `ECTimeUnit` and `ECTrigger` are not visible in the Java API. Instead, they are encapsulated by the `ECSpec` methods:

- `get/setDurationMillis`
- `get/setDurationReadCycles`
- `get/setStableSetIntervalMillis`
- `get/setStableSetIntervalReadCycles`
- `get/setRepeatPeriodMillis`
- `get/setStartTrigger`
- `get/setStopTrigger`

See the Javadoc for information on these methods.

XML: To express `duration` and `stableSetInterval` in read cycles (rather than milliseconds), use the `ECBoundarySpec` elements `durationReadCycles` and `stableSetIntervalReadCycles`.

ECTime

`ECTime` denotes a span of time measured in physical time units. Used to specify time values in `ECBoundarySpec`. See ECBoundarySpec on page 5-7 and ECBoundarySpec Implementation Notes on page 5-9.

EPCglobal ALE

```
duration : long
unit : ECTimeUnit
---
```

ECTimeUnit

`ECTimeUnit` is an enumerated type denoting different units of physical time that may be used in an `ECBoundarySpec`. See ECBoundarySpec on page 5-7 and ECBoundarySpec Implementation Notes on page 5-9. `ECTimeUnit` currently supports only one time unit (milliseconds).

EPCglobal ALE

```
<<Enumerated Type>>
MS          // Milliseconds
```

ECTrigger

ECTrigger denotes a URI that is used to specify a start or stop trigger for an event cycle. See ECBoundarySpec on page 5-7 and ECBoundarySpec Implementation Notes on page 5-9.

EPCglobal ALE

```
trigger : URI
---
```

ECReportSpec

Java implementation package: `com.connecterra.ale.api`

An `ECReportSpec` specifies one report to be returned from executing an event cycle. An `ECSpec` may contain one or more `ECReportSpec` instances.

EPCglobal ALE

```
reportName : string
reportSet : ECReportSetSpec
filter : ECFilterSpec
group : ECGroupSpec
output : ECReportOutputSpec
reportIfEmpty : boolean
reportOnlyOnChange : boolean
```

RFTagAware Extensions

```
essential : boolean
applicationData : string
---
```

The `reportSet` parameter specifies what set of EPCs is considered for reporting: all currently read, additions from the previous event cycle, or deletions from the previous event cycle. The `filter` parameter specifies how the raw EPCs are filtered before inclusion in the report. The `group` parameter (of type `ECGroupSpec`) specifies how the filtered EPCs are grouped together for reporting.

If no `group` parameter is specified, then all EPCs are placed in a single default group. The `output` parameter specifies whether to return the EPCs themselves, a count, or both.

The `reportIfEmpty` parameter specifies whether this report should be included in the final `ECReports` instance if the final, filtered list of EPCs is empty (i.e., if the final EPC list would be empty, or if the final count would be zero). If the parameter is set to:

- `false` – (default) This report is omitted when empty.
- `true` – This report is always included, even if it is empty.

If `reportOnlyOnChange` set to true, in the case of a standing report request, then reports will not be sent to subscribers unless the filtered list of EPCs is different than the previous event cycle's filtered list of EPCs. This comparison takes place before the filtered list has been modified based on `reportSet` or `output` parameters. The comparison also disregards whether the previous report was actually sent due to the effect of this boolean, or the `reportIfEmpty` boolean.

The `essential` parameter specifies whether this report is considered "essential" for the containing event cycle. "Essential" means that this report must be present for an `ECReports` instance to be generated at all. In the event that more than one report is marked "Essential", all such reports must be present for an ECReports instance to be generated.

For example, in a shipment receiving application there may be an event cycle with two `ECReportSpec` instances. The first, for which `essential=true`, has a `filter` set to include only those tags that match the tags expected in a particular shipment. The second, for which `essential=false`, has a count of all tags read. If a shipment is received that contains at least one item on the expected list, then an `ECReports` instance is delivered that contains a list of the tags expected, and a total count of all tags read. If, however, a shipment contains no tags from the expected list, the `essential` setting on the first report will suppress the generation of any report at all, and no notification will be delivered.

> **Note:** If the report has been marked `essential=true`, consider whether to change the default "empty report" behavior (omit the report when it is empty) controlled by the `reportIfEmpty` parameter.

The `reportName` parameter is an arbitrary string that is copied to the `ECReport` instance created when this event cycle completes. The purpose of the `reportName` parameter is so that clients can distinguish which of the `ECReport` instances it receives corresponds to which `ECReportSpec` instance contained in the original `ECSpec`. This is especially useful in cases where fewer reports are delivered than there were `ECReportSpec` instances in the `ECSpec`, because false `reportIfEmpty` settings suppressed the generation of some reports.

ECReportSpec Implementation Notes

Java: `ECFilterSpec`, `ECGroupSpec`, and `ECReportOutputSpec` are not visible in the Java API. Instead, they are encapsulated in the `ECReportSpec` methods:

- `get/set/addIncludePattern` and `get/set/addExcludePattern`

  See ECFilterSpec on page 5-12.

- `get/setGroupSpec`

  See ECGroupSpec on page 5-13.

- `includeList`, `includeCount`

  See ECReportOutputSpec on page 5-16.

ECReportSetSpec

Java implementation package: `com.connecterra.ale.api`

`ECReportSetSpec` is an enumerated type denoting what set of EPCs is to be considered for filtering and output: all EPCs read in the current event cycle, additions from the previous event cycle, or deletions from the previous event cycle.

```
<<Enumerated Type>>
CURRENT
ADDITIONS
DELETIONS
```

ECFilterSpec

An `ECFilterSpec` specifies what EPCs are to be included in the final report.

```
includePatterns : List    // List of URI-formatted EPC patterns
excludePatterns : List    // List of URI-formatted EPC patterns
```

The `ECFilterSpec` implements a flexible filtering scheme based on two pattern lists. Each list contains zero or more URI-formatted EPC patterns. Each EPC pattern denotes a single EPC, a range of EPCs, or some other set of EPCs. (Patterns are described in detail below in EPC Patterns on page 5-13.)

An EPC is included in the final report if (a) the EPC does not match any pattern in the `excludePatterns` list, and (b) the EPC does match at least one pattern in the `includePatterns` list. The (b) test is omitted if the `includePatterns` list is empty.

This can be expressed in mathematical notation as follows:

```
F(R) = { epc  |  epc in R & epc in I1 & … & epc in In & epc not in E1 & … & epc
not in En }
```

where `Ii` denotes the set of EPCs matched by the i^{th} pattern in the `includePatterns` list, and `Ei` denotes the set of EPCs matched by the i^{th} pattern in the `excludePatterns` list.

Java Implementation Notes: `ECFilterSpec` is not visible in the Java API. Instead, it is encapsulated by the `ECReportSpec` methods:

- `get/set/addIncludePattern`
- `get/set/addExcludePattern`

See the Javadoc for information on these methods.

EPC Patterns

EPC Patterns are used to specify filters within an `ECFilterSpec`. The complete syntax is defined by the *EPCglobal Tag Data Standards*, Version 1.1 Revision 1.27. Please consult that document (available at http://www.epcglobalinc.org/standards_technology/specifications.html) for full details; highlights are summarized here.

A single EPC pattern is a URI-formatted string that denotes a single EPC or set of EPCs. The general format is:

```
urn:epc:pat:TagEncodingName:Filter.DomainManager.ObjectClass.SerialNumber
```

where the four fields `Filter`, `DomainManager`, `ObjectClass`, and `SerialNumber` correspond to fields of an EPC. (Depending on the `TagEncodingName`, some of these fields may not actually be present. Consult the *EPCglobal Tag Data Standards* for details.) In an EPC pattern, each of those fields may be (a) a decimal integer, meaning that a matching EPC must have that specific value in the corresponding field; (b) an asterisk (`*`), meaning that a matching EPC may have any value in that field; or (c) a range denoted like `[lo-hi]`, meaning that a matching EPC must have a value between the decimal integers `lo` and `hi`, inclusive. (The tag data standards document includes restrictions and further details not documented here.)

Here are some examples. In these examples, assume that 20 is the Domain Manager for XYZ Corporation, and 300 is the Object Class for its UltraWidget product, and that GID-96 tag encodings are used.

`urn:epc:pat:gid-96:20.300.4000`	Matches the tag for UltraWidget serial number 4000.
`urn:epc:pat:gid-96:20.300.*`	Matches any UltraWidget, regardless of serial number.
`urn:epc:pat:gid-96:20.*.[5000-9999]`	Matches any XYZ Corporation product whose serial number is between 5000 and 9999, inclusive.
`urn:epc:pat:gid-96:*.*.*`	Matches any GID-96.

ECGroupSpec

`ECGroupSpec` defines how filtered EPCs are grouped together for reporting.

```
patternList : List
// List of pattern URIs
---
```

For detailed information, see:

- About Group Reports (page 5-14)

Java Implementation Notes: `ECGroupSpec` is not visible in the Java API. Instead, it is encapsulated by the `ECReportSpec` methods `getGroupSpec` and `setGroupSpec`.

See the Javadoc for information on these methods.

About Group Reports

Sometimes it is useful to group EPCs read during an event cycle based on portions of the EPC or attributes of the objects identified by the EPCs. For example, in a shipment receipt verification application, it is useful to know the quantity of each type of case (for example, each distinct case GTIN), but not necessarily the serial number of each case. This requires slightly more complex processing, based on grouping patterns.

You specify groups by supplying one or more group patterns in the `patternList` field of `ECGroupSpec`. Each element of the pattern list is an EPC Pattern URI as defined by the *EPCglobal Tag Data Standards*, extended by allowing the character X in each position where a * character is allowed. Pattern URIs used in an `ECGroupSpec` are interpreted as follows:

Pattern URI Field	Meaning
X	Create a different group for each distinct value of this field.
*	All values of this field belong to the same group.
Number	Only EPCs having *Number* in this field will belong to this group.
[Lo-Hi]	Only EPCs whose value for this field falls within the specified range will belong to this group.

Here are examples of pattern URIs used as grouping patterns:

Pattern URI	Meaning
`urn:epc:pat:sgtin-64:X.*.*.*`	Groups by filter value (for example, case/pallet).
`urn:epc:pat:sgtin-64:*.X.*.*`	Groups by company prefix.
`urn:epc:pat:sgtin-64:*.X.X.*`	Groups by company prefix and item reference (groups by specific product).
`urn:epc:pat:sgtin-64:X.X.X.*`	Groups by company prefix, item reference, and filter.
`urn:epc:pat:sgtin-64:3.X.*.[0-100]`	Creates a different group for each company prefix, including in each such group only EPCs having a filter value of 3 and serial numbers in the range 0 through 100, inclusive.

In the corresponding `ECReport`, each group is named by another EPC Pattern URI that is identical to the grouping pattern URI, except that the group name URI has an actual value in every position where the grouping pattern URI had an X character.

For example, if these are the filtered EPCs read for the current event cycle:

```
urn:epc:tag:sgtin-64:3.0036000.123456.400
urn:epc:tag:sgtin-64:3.0036000.123456.500
urn:epc:tag:sgtin-64:3.0029000.111111.100
urn:epc:tag:sscc-64:3.0012345.31415926
```

Then a pattern list consisting of just one element, like this:

```
urn:epc:pat:sgtin-64:*.X.*.*
```

would generate the following groups in the report:

Group Name	EPCs in Group
urn:epc:pat:sgtin-64:*.0036000.*.*	urn:epc:tag:sgtin-64:3.0036000.123456.400 urn:epc:tag:sgtin-64:3.0036000.123456.500
urn:epc:pat:sgtin-64:*.0029000.*.*	urn:epc:tag:sgtin-64:3.0029000.111111.100
[default group]	urn:epc:tag:sscc-64:3.0012345.31415926

Every filtered EPC that is part of the event cycle is part of exactly one group. If an EPC does not match any of the EPC Pattern URIs in the pattern list, it is included in a special "default group." The name of the default group is null. In the above example, the SSCC EPC did not match any pattern in the pattern list, and so was included in the default group.

As a special case of the above rule, if the pattern list is empty (or if the group parameter of the `ECReportSpec` is null or omitted), then all EPCs are part of the default group.

In order to insure that each EPC is part of only one group, there is an additional restriction that all patterns in the pattern list must be pairwise disjoint. Disjointness of two patterns is defined as follows. Let `Pat_i` and `Pat_j` be two pattern URIs, written as a series of fields as follows:

```
Pat_i = urn:epc:pat:type_i:field_i_1.field_i_2.field_i_3...
Pat_j = urn:epc:pat:type_j:field_j_1.field_j_2.field_j_3...
```

Then `Pat_i` and `Pat_j` are disjoint if:

- `type_i` is not equal to `type_j`
- `type_i` = `type_j` but there is at least one field `k` for which `field_i_k` and `field_j_k` are disjoint, as defined by the table below:

	X	*	Number	[Lo-Hi]
X	Not disjoint	Not disjoint	Not disjoint	Not disjoint
*****	Not disjoint	Not disjoint	Not disjoint	Not disjoint
Number	Not disjoint	Not disjoint	Disjoint if the numbers are different	Disjoint if the number is not included in the range
[Lo-Hi]	Not disjoint	Not disjoint	Disjoint if the number is not included in the range	Disjoint if the ranges do not overlap

The formal definition of grouping is as follows. A group operator G is specified by a list of pattern URIs:

```
G = (Pat_1, Pat_2, ..., Pat_N)
```

Let each pattern be written as a series of fields, where each `field_i_j` is either X, *, *Number*, or [*Lo-Hi*].

```
Pat_i = urn:epc:pat:type_i:field_i_1.field_i_2.field_i_3...
```

Then the definition of `G(epc)`, the group name associated with a specific EPC, is as follows. Let `epc` be written like this:

```
urn:epc:tag:type_epc:field_epc_1.field_epc_2.field_epc_3...
```

The `epc` is said to match `Pat_i` if

- *type_epc* = *type_i;* and
- For each field k, one of the following is true:
  - *field_i_k* = X
  - *field_i_k* = *
  - *field_i_k* is a number, equal to *field_epc_k*
  - *field_i_k* is a range [*Lo-Hi]*, and *Lo* is less than or equal to *field_epc_k*, which is less than or equal to *Hi*

Because of the disjointedness constraint specified above, the epc is guaranteed to match at most one of the patterns in `G`.

The group name `G(epc)` is then defined as follows:

- If `epc` matches `Pat_i` for some `i`, then

  `G(epc) = urn:epc:pat:type_epc:field_g_1.field_g_2.field_g_3...`

  where for each *k*, *field_g_k* = *, if *field_i_k* = *; or *field_g_k* = *field_epc_j*, otherwise.
- If `epc` does not match `Pat_i` for any `i`, then `G(epc)` = the default group.

ECReportOutputSpec

`ECReportOutputSpec` specifies how the final set of EPCs is to be reported.

```
includeEPC : boolean
includeTag : boolean
includeRawHex : boolean
includeRawDecimal : boolean
includeCount : boolean
---
```

If any of the four booleans `includeEPC`, `includeTag`, `includeRawHex`, or `includeRawDecimal` is true, the report includes a list of the EPCs in the final set for each group. Each element of this list, when included, includes the formats specified by these four booleans. If `includeCount` is true, the report includes a count of the EPCs in the final set for each group. Both may be true, in which case each group includes both a list and a count. If all five booleans `includeEPC`, `includeTag`, `includeRawHex`, `includeRawDecimal`, `and includeCount` are false, then the `define` and `immediate` methods raise an exception.

Java Implementation Notes: `ECReportOutputSpec` is not visible in the Java API. Instead, it is encapsulated by these `ECReportSpec` methods:

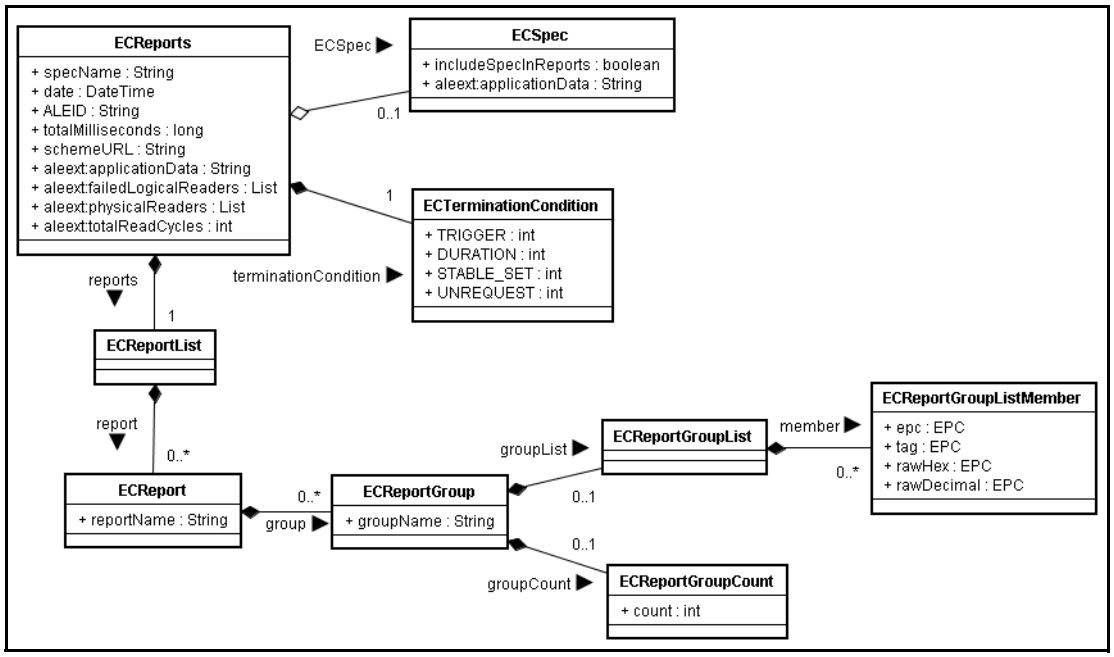
- `setIncludeEPC`
- `setIncludeTag`
- `setIncludeRawHex`
- `setIncludeRawDecimal`
- `setIncludeCount`

See the Javadoc for information on these methods.

ECReports

Java implementation package: `com.connecterra.ale.api`

ECReports UML Diagram



`ECReports` is the output from an event cycle.

EPCglobal ALE

```
specName : string
date : dateTime
ALEID : string
totalMilliseconds : long
terminationCondition : ECTerminationCondition
spec : ECSpec
reports : List // List of ECReport instances
schemaURL : URI
```

RFTagAware Extensions

```
totalReadCycles : int
applicationData : string
physicalReaders : List      // List of strings, each naming a physical reader
failedLogicalReaders : List // List of strings, each naming a logical reader
```

The "meat" of an `ECReports` instance is the list of `ECReport` instances, each corresponding to an `ECReportSpec` instance in the event cycle's `ECSpec`. In addition to the reports themselves, `ECReports` contains a number of "header" fields that provide useful information about the event cycle:

Field	Description
`specName`	The name of the `ECSpec` that controlled this event cycle. In the case of an `ECSpec` that was requested using the `immediate` method, this name is one chosen by the Edge Server.
`date`	The date and time when the event cycle ended.
`ALEID`	An identifier for the deployed instance of the Edge Server. This value is set by the `ale.savantID` property in the `edge.props` file and can be set in the installer.
`totalMilliseconds`	The total time, in milliseconds, from the start of the event cycle to the end of the event cycle.
`terminationCondition`	Indicates what kind of event caused the event cycle to terminate: the receipt of an explicit stop trigger, the expiration of the event cycle duration, or the set of EPCs being stable for the prescribed amount of time. These correspond to the possible ways of specifying the end of an event cycle as defined in ECBoundarySpec on page 5-7.
`spec`	A copy of the ECSpec that generated this ECReports instance. Only included if the ECSpec has `includeSpecInReports` set to true.
`schemaURL`	Specifies a URL where the XML schema for ALE API as used in this version of RFTagAware may be found. This schema includes RFTagAware extensions to the EPCglobal ALE specification.

Field	Description
`totalReadCycles`	The total time, in read cycles, from the start of the event cycle to the end of the event cycle. When more than one reader contributed read cycles to this event cycle, this number is the number of read cycles contributed by the reader that contributed the fewest number of read cycles.
`applicationData`	A copy of the `applicationData` field of the `ECSpec` that controlled this event cycle.
`physicalReaders`	A list of strings, each identifying one of the physical readers that contributed to this event cycle. The mapping of physical and logical reader names is specified in the `edge.props` file.
`failedLogicalReaders`	A list of strings, each identifying a logical reader that reported failures during this event cycle. This list is always a subset of the `logicalReaders` field of the `ECSpec` that controlled this event cycle. If no failures occurred, `failedLogicalReaders` is empty.

ECTerminationCondition

Java implementation package: `com.connecterra.ale.api`

`ECTerminationCondition` is an enumerated type that describes how an event cycle was ended.

```
<<Enumerated Type>>
EPCglobal ALE
TRIGGER
DURATION
STABLE_SET
UNREQUEST
```

The first three values, `TRIGGER`, `DURATION`, and `STABLE_SET`, correspond to the receipt of an explicit stop trigger, the expiration of the event cycle `duration`, or the set of EPCs being stable for the event cycle `stableSetInterval`, respectively. These are the possible stop conditions described in ECBoundarySpec on page 5-7.

The last value, `UNREQUEST`, corresponds to an event cycle being terminated because there were no longer any clients requesting it. By definition, this value cannot actually appear in an `ECReports` instance sent to any client.

ECReport

Java implementation package: `com.connecterra.ale.api`

`ECReport` represents a single report within an `ECReports` instance generated by an event cycle.

EPCglobal ALE

```
reportName : string
groups : List // List of ECReportGroup instances
```

RFTagAware Extensions

```
applicationData : string
---
```

The `reportName` field is a copy of the `reportName` field from the corresponding `ECReportSpec` within the `ECSpec` that controlled this event cycle. The `groups` field is a list containing one element for each group in the report as controlled by the `group` field of the corresponding `ECReportSpec`. When no grouping is specified, the groups list contains the single default group. Each element of the list is an instance of `ECReportGroup`.

Java Implementation Notes: The Java API provides two methods (`hasCount` and `hasList`) that indicate whether each contained `ECReportGroup` instance includes an `ECReportGroupCount` instance and an `ECReportGroupList` instance, respectively.

ECReportGroup

Java implementation package: `com.connecterra.ale.api`

`ECReportGroup` represents one group within an `ECReport`.

EPCglobal ALE

```
groupName : string
groupList : ECReportGroupList
groupCount : ECReportGroupCount
---
```

The `groupName` is null for the default group (in XML, the group name is omitted to indicate the default group). Otherwise, the `groupName` is a string calculated as specified in About Group Reports on page 5-14.

The `groupList` field is null if the `includeEPC`, `includeTag`, `includeRawHex`, and `includeRawDecimal` fields of the corresponding `ECReportOutputSpec` are all false.

The `groupCount` field is null if the `includeCount` field of the corresponding `ECReportOutputSpec` is false.

Java Implementation Notes: The Java API provides two methods (`hasCount` and `hasList`) that indicate whether this `ECReportGroup` instance includes an `ECReportGroupCount` instance and an `ECReportGroupList` instance, respectively.

ECReportGroupList

An `ECReportGroupList` is included in an `ECReportGroup` when any of the four boolean fields `includeEPC`, `includeTag`, `includeRawHex`, and `includeRawDecimal` of the corresponding `ECReportOutputSpec` is true.

EPCglobal ALE

```
members : List // List of EPCReportGroupListMember instances
---
```

The order in which EPCs are enumerated within the list is unspecified.

Java Implementation Notes: `ECReportGroupList` is not visible in the Java API. Instead, it is encapsulated by the `ECReportGroup` method `getGroupList`. See the Javadoc for information on this method.

ECReportGroupListMember

Java implementation package: `com.connecterra.ale.api`

Each member of the `ECReportGroupList` is an `ECReportGroupListMember` as defined below. The reason for having `ECReportGroupListMember` is to allow multiple EPC formats to be included, and to provide an extension point for adding per-EPC information to the list report.

EPCglobal ALE

```
epc : URI
tag : URI
rawHex : URI
rawDecimal : URI
---
```

Each of these fields either contains a URI as described below or is null, depending on the value of a boolean in the corresponding `ECReportOutputSpec`. For example, the `epc` field is non-null if and only if the `includeEPC` field of `ECReportOutputSpec` is true.

When non-null, the `epc` field contains an EPC represented as a pure identity URI according to the *EPCglobal Tag Data Standards* (`urn:epc:id:`...). A pure identity URI contains just the EPC, with no additional information such as tag type, filter bits, etc. If the information on the tag cannot be successfully decoded into a pure identity URI, the `epc` field contains a raw decimal URI instead.

When non-null, the `tag` field contains an EPC represented as a tag URI according to the *EPCglobal Tag Data Standards* (`urn:epc:tag:`...). A tag URI contains all information on the tag, including the EPC, tag type, and filter bits (when applicable). The tag URI is also suitable for use in writing tags using the ALEPC API (see Chapter 2: Reading and Writing Tags and Chapter 6: Writing Tags Using

the ALE API). If the information on the tag cannot be successfully decoded into a tag URI, the `tag` field contains a raw decimal URI instead.

When non-null, the `rawDecimal` field contains a raw tag value represented as a raw decimal URI according to the *EPCglobal Tag Data Standards* (`urn:epc:raw:`…).

When non-null, the `rawHex` field contains a raw tag value represented as a raw hexadecimal URI according to the following extension to the *EPCglobal Tag Data Standards*. The URI is determined by concatenating the following: the string `urn:epc:raw:`, the length of the tag value in bits, a dot (.) character, a lowercase x character, and the tag value considered as a single hexadecimal integer. The length value preceding the dot character has no leading zeros. The hexadecimal tag value following the dot has a number of characters equal to the length of the tag value in bits divided by four and rounded up to the nearest whole number, and uses only uppercase letters for the hexadecimal digits A, B, C, D, E, and F.

Each distinct tag value included in the report has a distinct `ECReportGroupListMember` element in the `ECReportGroupList`, even if those `ECReportGroupListMember` elements would be identical due to the formats selected. In particular, it is possible for two different tags to have the same pure identity EPC representation; for example, two SGTIN-64 tags that differ only in the filter bits. If both tags are read in the same event cycle, and `ECReportOutputSpec` specified `includeEPC` true and all other formats false, then the resulting `ECReportGroupList` has two `ECReportGroupListMember` elements, each having the same pure identity URI in the `epc` field. In other words, the result should be equivalent to performing all duplicate removal, additions/deletions processing, grouping, and filtering before converting the raw tag values into the selected representation(s).

The situation in which this rule applies is expected to be extremely rare. In theory, no two tags should be programmed with the same pure identity, even if they differ in filter bits or other fields not part of the pure identity.

See the *EPCglobal Tag Data Standards* for more information on URI representations of Electronic Product Codes.

ECReportGroupCount

An `ECReportGroupCount` is included in an `ECReportGroup` when the `includeCount` field of the corresponding `ECReportOutputSpec` is true.

EPCglobal ALE

```
count : int
---
```

The count field is the total number of distinct EPCs that are part of this group.

Java Implementation Notes: `ECReportGroupCount` is not visible in the Java API. Instead, it is encapsulated by the `ECReportGroup` method `getGroupCount`. See the Javadoc for information on this method.

Other ALE API Types

This section defines other types that are used in the RFTagAware ALE API:

These types are all RFTagAware extensions to the EPCglobal specification.

ECSpecInfo (RFTagAware Extension)

Java implementation package: `com.connecterra.ale.api`

`ECSpecInfo` gives information about the current state of a defined `ECSpec`.

RFTagAware Extensions

```
subscriberCount : int
activationCount : int
lastActivated : timestamp
lastReported : timestamp
isSuspended : boolean
---
```

Field	Description
`subscriberCount`	The number of current subscribers for this `ECSpec`.
`activationCount`	The number of times the `ECSpec` has transitioned into the active state since it was first defined.
`lastActivated`	When the `ECSpec` last transitioned into the active state.
`lastReported`	When the `ECSpec` last delivered a report to subscribers. This may be different than `lastActivated` because the settings in `ECReportSpecs` might cause the `ECSpec` not to deliver a report if no matching tags were read.
`isSuspended`	Indicates whether or not the `ECSpec` is in a suspended state.

XML Implementation Notes: `ECSpecInfo` values are expressed in the element `EventCycleSpecInfo`.

ECSubscriptionInfo (RFTagAware Extension)

Java implementation package: `com.connecterra.ale.api`

`ECSubscriptionInfo` gives information about a specific subscriber to an `ECSpec`.

```
controls : ECSubscriptionControls
consecutiveFailureCount : int
lastSuccessTime : timestamp
---
```

Field	Description
`controls`	The controls that govern when the subscription is automatically unsubscribed in case of delivery failures.
`consecutiveFailureCount`	The number of consecutive times that reports were unable to be delivered. This is zero if the most recent report was delivered successfully, one if the most recent report was not delivered but the one before that was, and so forth.
`lastSuccessTime`	The date and time a report was last successfully delivered.

XML Implementation Notes: `ECSubscriptionInfo` values are expressed in the element `EventCycleSubscriptionInfo`.

ECSubscriptionControls (RFTagAware Extension)

Java implementation package: `com.connecterra.ale.api`

`ECSubscriptionControls` contains parameters that govern when subscriptions are automatically unsubscribed in case of delivery failures. Most subscriptions are governed by a default set of parameters that are configured when the Edge Server is deployed (see the *RFTagAware Deployment Guide*). When a client wants to override these settings for a specific subscription, the client may use the form of the ALE `subscribe` call that takes an explicit `ECSubscriptionControls` argument.

```
failureLimitCount : int
failureLimitInterval : long
---
```

Field	Description
`failureLimitCount`	The maximum number of failed notification deliveries before a subscription is unsubscribed.
`failureLimitInterval`	The maximum interval of time (in milliseconds) during which notification delivery may fail before a subscription is unsubscribed.

XML Implementation Notes: `ECSubscriptionControl` values are expressed in the element `EventCycleSubscriptionControls`.

XML Representations

The focal points of the ALE tag reading interface from an application's perspective are the `ECSpec` and `ECReports` objects. The Edge Server provides a standard way of representing `ECSpec` and `ECReports` instances in XML. The XML form of `ECReports` is used by most of the asynchronous event cycle delivery mechanisms, as described in Chapter 3: Asynchronous Notification Mechanisms. User applications may also find the XML forms very useful as a means of interchange, and for persistent storage.

The XML forms of `ECSpec` and `ECReports` are defined by the XSD files:

- `EPCglobal-ale-1_0.xsd`

  Defines EPCglobal ALE schema; references RFTagAware extensions. `ECSpec` and `ECReports` are both defined in this schema. The top-level element for `ECSpec` is `ECSpec`; for `ECReports` the top-level element is `ECReports`.

- `EPCglobal.xsd`

  Defines the EPCglobal common types, `Document` and `EPC`, referred to by `EPCglobal-ale-1_0.xsd`.

- `EPCglobal-ale-1_0-RFTagAware-extensions.xsd`

  Defines the RFTagAware schema extensions.

These files are located in your RFTagAware installation directory under `share/schemas`.

The Java binding for ALE provides XML serializer and deserializer classes for translating between the XML representation and the Java representation of the `ECSpec` and `ECReports` types. Applications may use these facilities to process reports received via the Edge Server's asynchronous event cycle delivery mechanisms, and for other purposes. The sample applications bundled with RFTagAware illustrate the use of the serializer and deserializer classes. See Using XML Serializers and Deserializers from Java (page 5-28) for more information.

The remainder of this section presents examples of `ECSpec` and `ECReports` as rendered into XML. These examples include additional line breaks and whitespace for the sake of readability. RFTagAware permits (but does not require) this whitespace when reading XML; usually RFTagAware omits this whitespace when writing XML.

ECSpec - Example

Here is an example `ECSpec` rendered into XML:

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

<ale:ECSpec xmlns:ale="urn:epcglobal:ale:xsd:1"
            xmlns:aleext="http://schemas.connecterra.com/EPCglobal-extensions/ale"
            creationDate="2004-11-15T16:18:43.500Z"
            schemaVersion="1.0"
```

```
            includeSpecInReports="false" >

<logicalReaders>
   <logicalReader>ConnecTerra1</logicalReader>
</logicalReaders>

<boundarySpec>
   <duration unit="MS">2000</duration>
</boundarySpec>

<reportSpecs>
   <reportSpec reportName="SubscribeSample Report">
   <reportSet set="CURRENT" />
   <output includeCount="true"
                   includeEPC="false"
                   includeRawDecimal="false"
                   includeRawHex="false"
                   includeTag="true"  />
   </reportSpec>
</reportSpecs>

<aleext:applicationData>application-specific data here</aleext:applicationData>

</ale:ECSpec>
```

ECReports - Example

Here is an example `ECReports` rendered into XML:

```
<ale:ECReports ALEID="EdgeServerID" creationDate="2005-01-06T16:47:57.296Z"
date="2005-01-06T16:47:57.296Z"
    schemaURL="http://schemas.connecterra.com/EPCglobal/ale-1_0.xsd"
    schemaVersion="1"
    specName="sampleECSpec"
    terminationCondition="DURATION"
    totalMilliseconds="2015"
    xmlns:ale="urn:epcglobal:ale:xsd:1"
    xmlns:aleext="http://schemas.connecterra.com/EPCglobal-extensions/ale">

<reports>
  <report reportName="SubscribeSample Report">
   <group>
    <groupList>
     <member>
      <tag>urn:epc:tag:gid-64-i:10.50.5</tag>
     </member>
     <member>
      <tag>urn:epc:tag:gid-64-i:10.40.4</tag>
     </member>
     <member>
      <tag>urn:epc:tag:gid-64-i:10.10.1</tag>
     </member>
     <member>
      <tag>urn:epc:tag:gid-64-i:10.30.3</tag>
     </member>
     <member>
      <tag>urn:epc:tag:gid-64-i:10.70.7</tag>
```

```
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.20.2</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.60.6</tag>
      </member>
     </groupList>
     <groupCount>
      <count>7</count>
     </groupCount>
    </group>
  </report>
</reports>

<aleext:applicationData>application-specific data here</aleext:applicationData>

<aleext:failedLogicalReaders/>

<aleext:physicalReaders>
  <aleext:physicalReader>SimReadr</aleext:physicalReader>
</aleext:physicalReaders>

<aleext:totalReadCycles>8</aleext:totalReadCycles>

</ale:ECReports>
```

Using the ALE Tag Reading API from Java

When you use the Java binding of the ALE API, there are additional Java interfaces and classes available to you beyond what is described previously in this chapter. This section gives a brief introduction to those additional interfaces and classes. For full documentation, see the Javadoc that is included in the RFTagAware installation.

To use the ALE tag reading API from Java, you create an instance of the `SOAPALEClient` class provided in the `com.connecterra.ale.client` package. This class implements the `ALE` interface as described in ALE: Main Tag Reading Interface on page 5-3 and provides all of the methods described there. It also provides an additional method, `getALEFactory`, which returns a factory for creating instances of other types, described below. The `SOAPALEClient` interacts with an RFTagAware Edge Server over the network using SOAP over HTTP. When you construct an instance of `SOAPALEClient`, you provide a service URL for the Edge Server with which you wish to interact.

When using the `SOAPALEClient` class, you need to create instances of `ECSpec` and other types described in Chapter 5: Reading Tags Using the ALE API. The `ALEFactory` interface (in package `com.connecterra.ale.api`) provides methods for creating instances of those types. You obtain an instance of the `ALEFactory` interface by calling the `getALEFactory` method provided by the `SOAPALEClient` class. When passing arguments to methods of a specific `SOAPALEClient` instance, you must always use the factory instance provided by that `SOAPALEClient` instance.

Using XML Serializers and Deserializers from Java

The Java binding of the ALE API provides some additional utility classes for reading and writing XML representations of the data types used in the ALE API. With these classes, you can convert a Java object representation of a particular data type into XML ("serialization"), and likewise convert an XML representation of a particular data type back into a Java object ("deserialization"). The XML schemas may be found in your RFTagAware installation directory at:

```
/share/schemas/EPCglobal-ale-1_0.xsd
/share/schemas/EPCglobal.xsd
/share/schemas/EPCglobal-ale-1_0-RFTagAware-extensions.xsd
```

To read and write XML for types used in the ALE tag reading API, you use an instance of the `XMLSerializationFactory` provided in the `com.connecterra.ale.encoding` package. There is only one static instance of this class, which you obtain using the static method `getInstance()`, passing the argument `XMLSerializationSyntax.EPCglobal_ale_1_0`. Using the `XMLSerializationFactory` instance, you can create instances of `XMLSerializer` and `XMLDeserializer` to serialize and deserialize instances of the following classes: `ECSpec`, `ECReports`, `ECSpecInfo`, `ECSubscriptionInfo`, and `ECSubscriptionControls`.

Chapter 6: Writing Tags Using the ALE API

Contents

This chapter describes the ALE API programming components you use to write tags.

Introduction to the ALE API Specification

This section provides a formal, abstract specification of the ALE API for writing tags. The external interface is defined by the ALEPC interface (See ALEPC: Main Tag Writing Interface on page 6-3). This interface makes use of a number of complex data types that are documented in the sections starting at PCSpec on page 6-5.

The general interaction model is that there are one or more clients that make method calls to the ALEPC interface. Each method call is a request, which causes the ALE engine to take some action and return results. Thus, methods of the ALEPC interface are synchronous.

The ALEPC interface also provides a way for clients to subscribe to events that are delivered asynchronously. This is done through methods that take a URI as an argument. Such methods return immediately, but subsequently the ALE engine within the Edge Server may asynchronously deliver information to the consumer denoted by the URI argument.

In the sections below, the API is described using UML class diagram notation, as shown below:

```
dataMember1 : Type1
dataMember2 : Type2
---
method1(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
method2(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
```

The box as a whole refers to a conceptual class, having the specified data members and methods.

The ALE API is realized in several equivalent forms within RFTagAware:

- There is a binding of the ALE API to Java, in which it takes the form of a collection of Java interface and class definitions.
- There is another binding of the ALE API to a SOAP web service, described by a WSDL file.
- The complex data types have a standard representation as XML documents, defined by an XSD schema.

Each of these concrete forms of the ALE API has a slightly different structure and gives slightly different names to the different conceptual classes, data members, and methods defined in UML within this section. This is unavoidable, owing to syntactic constraints and stylistic norms within these different implementation technologies.

In most cases, the mapping from conceptual UML to the concrete details of any particular binding is very straightforward; where it is not, the specific documentation for each binding will make clear the relationship to the UML. The UML-level descriptions in this section should be considered normative.

- For specifics of the Java binding, see the Javadoc that is included in the RFTagAware installation.

- For specifics of the WSDL binding, see the WSDL file that is included in your RFTagAware installation directory at:
  `share/schemas/ALEPCService.wsdl`
- For specifics of the XML representation of the complex data types, see the XSD file that is included in your RFTagAware installation directory at:
  `share/schemas/ALEPC.xsd`

  See also XML Representations on page 6-12.

ALEPC: Main Tag Writing Interface

`ALEPC` is the main interface for programming tags. The term "tag programming" refers to the act of causing an EPC value to become associated with some physical entity, such as an RFID tag or a printed label.

Java implementation package: `com.connecterra.alepc.api`

```
---
define(specName:String, spec:PCSpec) : void
redefine(specName:String, spec:PCSpec) : void
suspend(specName:String) : void
unsuspend(specName:String) : void
undefine(specName:String) : void
get(specName:String) : PCSpec
getPCSpecInfo(specName:String) : PCSpecInfo
listPCSpecNames() : List
subscribe(specName:String, uri:URI, controls:PCSubscriptionControls) : void
unsubscribe(specName:String, uri:URI) : void
listSubscribers(specName:String) : List
getPCSubscriptionInfo(specName:String,subscriber:URI) : PCSubscriptionInfo
poll(specName:String, epcVal:URI) : PCWriteReport
immediate(spec:PCSpec, epcVal:URI) : PCWriteReport
defineEPCCache(cacheName:String, spec:EPCCacheSpec,
replenishment:EPCPatterns) : void
redefineEPCCache(cacheName:String, newSpec:EPCCacheSpec) : void
undefineEPCCache(cacheName:String) : EPCPatterns
getEPCCache(cacheName:String) : EPCCacheSpec
getEPCCacheSpecInfo(cacheName:String, includeCacheContent:boolean) :
EPCCacheSpecInfo
listEPCCacheSpecNames() : List
replenishEPCCache(cacheName:String, replenishment:EPCPatterns) : void
depleteEPCCache(cacheName:String) : EPCPatterns
subscribeEPCCache(cacheName:String, uri:URI,
controls:PCSubscriptionControls) : void
```

```
unsubscribeEPCCache(cacheName:String, uri:URI) : void
listEPCCacheSubscribers(cacheName:String) : List
getEPCCacheSubscriptionInfo(cacheName:String, subscriber:URI) : PCSubscriptionInfo
listLogicalReaderNames() : List
```

Field	Description
`define`	Define a new programming cycle specification for use with the `poll` and `subscribe` methods.
`defineEPCCache`	Define an EPC cache that can be used by programming cycles to obtain EPC values for programming operations.
`depleteEPCCache`	Cause the indicated EPC cache to become depleted (empty).
`get`	Look up and return a previously defined programming cycle specification by name.
`getEPCCache`	Look up and return a previously defined EPC cache specification by name.
`getEPCCacheSpecInfo`	Return administrative information about an EPC cache.
`getEPCCacheSubscriptionInfo`	Return administrative information about an EPC cache subscriber.
`getPCSpecInfo`	Return administrative information about a programming cycle specification.
`getPCSubscriptionInfo`	Return administrative information about a programming cycle subscriber.
`immediate`	Immediately define a programming cycle specification and activate it for one programming cycle, synchronously returning a report.
`listEPCCacheSpecNames`	Return a list of the names of all EPC caches currently defined.
`listEPCCacheSubscribers`	Return a list of URIs that are subscribed to asynchronous reports for the specified EPC cache name.
`listLogicalReaderNames`	Return a list of all logical reader names that can be used for programming.
`listPCSpecNames`	Return a list of the names of all programming cycle specifications currently defined.
`listSubscribers`	Return a list of URIs that are subscribed to asynchronous reports for the specified `PCSpec` name.
`poll`	Activate a previously defined programming cycle specification for one programming cycle, synchronously returning a report.
`redefine`	Replace the `PCSpec` for a programming cycle with a new `PCSpec`.
`redefineEPCCache`	Replace the `EPCCacheSpec` having the specified name with a new `EPCCacheSpec`. All subscriptions and other metadata remain unchanged, and the cache contents are not altered.
`replenishEPCCache`	Append a set of EPC pattern URIs to the indicated EPC cache.
`subscribe`	Subscribe to asynchronous report delivery from a programming cycle specification.
`subscribeEPCCache`	Subscribe to asynchronous report delivery from an EPC cache.
`suspend`	Suspend the named programming cycle.
`undefine`	Undefine a programming cycle specification.

Field	Description
`undefineEPCCache`	Undefine an EPC cache.
`unsubscribe`	Unsubscribe a specified destination from receiving asynchronous delivery of reports from a specified programming cycle specification.
`unsubscribeEPCCache`	Unsubscribe a specified destination from receiving asynchronous delivery of reports from a specified EPC cache.
`unsuspend`	Return a suspended programming cycle to its normal state.

PCSpec

Java implementation package: `com.connecterra.alepc.api`

A `PCSpec` is a complex type describing a programming cycle. A programming cycle is an interval of time during which a single tag is written and verified.

A `PCSpec` contains:

- A list of readers that should try to write the tag. Each member of this list may be a single logical reader, or the name of a composite reader. For information on composite readers, see Using Composite Readers on page 2-14.
- Optional start and stop triggers, which provide one way of starting and ending the programming cycle.
- How many times the reader(s) should try to write the tag. This can be expressed as a number of attempts (`trials`) or as a length of time (`duration`).
- Optional name of an EPC cache from which this programming cycle obtains EPC values. For information on EPC caches, see EPC Caches and Pools on page 2-8.

It also contains an optional "application data" string, which is simply copied unmodified into every `PCWriteReport` instance generated from this `PCSpec`.

For a narrative description programming cycles and their use of `PCSpec` instances, see Programming Cycles on page 2-6.

```
logicalReaders : List
readerParameters: Map
applicationData: string
cacheName: string
duration: long
startTrigger: URI
stopTrigger: URI
trials: int
---
```

Field	Description
`logicalReaders`	List of logical or composite reader names.
`readerParameters`	Maps parameter names to parameter values. This can be used to pass information to a reader. For example, a RFID label printer might define a reader parameter that it uses to obtain the text and graphics to be printed on labels. See the *RFTagAware Reader Configuration Guide* for information about the capabilities of specific reader and printer devices.
`applicationData`	A string that is copied unmodified into every `PCWriteReport` instance generated from this `PCSpec`.
`cacheName`	The name of the EPC cache from which this programming cycle obtains EPC values.
`duration`	The maximum amount of time to run EPC writing trials before failing a programming cycle.
`startTrigger`	Trigger that begins a programming cycle.
`stopTrigger`	Trigger that ends a programming cycle.
`trials`	Maximum number of EPC writing trials to run before failing a programming cycle.

PCSpecInfo

Java implementation package: `com.connecterra.alepc.api`

Describes administrative information for a `PCSpec`.

```
activationCount : int
cacheSize: long
lastActivated: long
lastReported: long
subscriberCount: int
isSuspended: boolean
---
```

Field	Description
`activationCount`	The number of times the programming cycle for the PCSpec has been activated since it was defined.
`cacheSize`	How many entries remain in the `PCSpec`'s associated EPC cache.
`lastActivated`	The last time the programming cycle for the `PCSpec` was activated.
`lastReported`	The last time a write report was generated by the programming cycle for the PCSpec.
`subscriberCount`	The number of URIs subscribed to a `PCSpec`.
`isSuspended`	True if the `PCSpec` processing is suspended.

PCSubscriptionControls

Java implementation package: `com.connecterra.alepc.api`

Describes how to handle failures in notification delivery. Used by `PCSubscriptionInfo` (see PCSubscriptionInfo on page 6-7) and `ALEPC.subscribe(String, URI, PCSubscriptionControls)` (see ALEPC: Main Tag Writing Interface on page 6-3).

```
failureLimitCount : int
failureLimitInterval : long
---
```

Field	Description
`failureLimitCount`	The maximum number of failed notification deliveries before a subscription is unsubscribed.
`failureLimitInterval`	The maximum interval of time notification delivery may fail before a subscription is unsubscribed.

PCSubscriptionInfo

Java implementation package: `com.connecterra.alepc.api`

Describes administrative information about a subscription.

```
consecutiveFailureCount : int
controls : PCSubscriptionControls
lastSuccessTime: long
---
```

Field	Description
`consecutiveFailureCount`	The number of failed notifications since the subscription was created, or since the last successful notification.
`controls`	The notification failure controls for this subscription. See PCSubscriptionControls on page 6-7.
`lastSuccessTime`	The absolute time in milliseconds of the most recent successful notification.

PCWriteReport

Java implementation package: `com.connecterra.alepc.api`

Report that describes a programming cycle's tag writing operation.

```
date : timestamp
savantID : string
specName : string
cacheName : string
applicationData : string
wasSuccessful : boolean
status : PCStatus
physicalReader : List
failedLogicalReaders : List
totalMilliseconds : long
totalTrials : int
cacheSize : long
EPC : URI
successfulLogicalReader : string
failureInfo : string
terminationCondition : PCTerminationCondition
---
```

Field	Description
`applicationData`	String that you set in the `PCSpec`. See PCSpec on page 6-5.
`cacheName`	The name of the EPC cache associated with this `PCSpec`.
`cacheSize`	How many EPC cache entries were left when the programming cycle completed.
`date`	The date and time the report was generated.
`EPC`	The EPC value that was written to the tag.
`failedLogicalReaders`	The logical readers that had some sort of failure during the programming cycle that generated this report.
`failureInfo`	Additional information about the failure, if available.
`physicalReader`	Names of physical readers that were involved in the programming cycle that generated this report.
`savantID`	The identifier for the Edge Server that generated this report.
`specName`	Name of the `PCSpec` that describes the just-completed programming cycle.
`status`	The status of the programming cycle. See PCStatus on page 6-9.
`successfulLogicalReader`	The logical reader that actually performed the successful tag write.
`terminationCondition`	The condition that terminated the failed programming cycle activation. See PCTerminationCondition on page 6-10.
`totalMilliseconds`	The total time in milliseconds during which the programming cycle was active.

Field	Description
totalTrials	The total number of trials for which the programming cycle was active.
wasSuccessful	True if the programming cycle succeeded. False if the programming cycle failed.

PCStatus

Java implementation package: com.connecterra.alepc.api

An enumerated type that identifies the termination status of a programming cycle.

```
<<Enumerated Type>>
SUCCESSFUL
NONE_IN_FIELD
NOT_WRITTEN
VERIFY_ERROR
MULTIPLE_IN_FIELD
LOCKED
INCOMPATIBLE_TAG_TYPE
READ_ONLY
CACHE_EMPTY
READER_BUSY
READER_ERROR
ENGINE_ERROR
```

Value	Description
CACHE_EMPTY	A programming cycle could not be started because the EPC cache was empty.
ENGINE_ERROR	The ALE engine itself has some kind of problem.
INCOMPATIBLE_TAG_TYPE	The tag (or reader) is a of a type that is not compatible with the EPC value that was supplied to be written to the tag (for example, a 96-bit EPC written to a 64-bit tag).
LOCKED	The tag is locked and therefore cannot be programmed.
MULTIPLE_IN_FIELD	Multiple tags were in the field of the programming cycle's reader(s).
NONE_IN_FIELD	No tags were in the field of the programming cycle's reader(s).
NOT_WRITTEN	The tag was not written (the verification readback yielded the original tag value).
READ_ONLY	The tag is a read-only type and therefore cannot be programmed.
READER_BUSY	One or more of the programming cycle's readers is already in use by a programming cycle or by an event cycle.
READER_ERROR	One or more of the programming cycle's readers has some kind of problem.
SUCCESSFUL	The programming cycle completed successfully.
VERIFY_ERROR	The tag was mis-programmed (the verification readback yielded a CRC error or value other than the intended one).

PCTerminationCondition

Java implementation package: `com.connecterra.alepc.api`

An enumerated type that describes the conditions that can cause a programming cycle to terminate with a failure

```
<<Enumerated Type>>
DURATION
FAILURE
TRIALS
TRIGGER
UNDEFINE
```

Value	Description
`DURATION`	The programming cycle was terminated due to exhausting the `duration` value specified in the `PCSpec`. A tag may still have been written.
`FAILURE`	The programming cycle was terminated due to a condition (such as multiple tags in field) for which retrying does not make sense.
`TRIALS`	The programming cycle was terminated due to exhausting the `trials` value specified in the `PCSpec`.
`TRIGGER`	The programming cycle was terminated due to receipt of a stop trigger. A tag may still have been written.
`UNDEFINE`	The programming cycle was terminated because the `PCSpec` was undefined or suspended.

EPCCacheSpec

Java implementation package: `com.connecterra.alepc.api`

Describes a tag cache.

```
applicationData: string
includeCacheContent : boolean
threshold : long
---
```

Field	Description
`applicationData`	String that will be returned in `EPCCacheReport` instances. See EPCCacheReport on page 6-11.

Field	Description
`includeCacheContent`	Indicates whether `EPCCacheReport` instances should include a description of the current cache contents (true) or just the count of the remaining cache entries (false).
`threshold`	Specifies a limit, such that when a cache's number of remaining EPC values drops to (or below) the limit, an `EPCCacheReport` is issued to subscribers. 0 means issue the `EPCCacheReport` when the EPC cache count drops to empty.

EPCCacheReport

Java implementation package: `com.connecterra.alepc.api`

Report that indicates that an EPC cache is low.

```
applicationData : string
cacheContent : EPCPatterns
cacheName : string
cacheSize : long
date : timestamp
savantID : string
threshold : long
---
```

Field	Description
`applicationData`	String that you set in the `EPCCacheSpec`. See EPCCacheSpec on page 6-10.
`cacheContent`	Describes the remaining content of the EPC cache. See EPCPatterns on page 6-12.
`cacheName`	The name of the EPC cache that this report describes.
`cacheSize`	How many EPC cache entries remain.
`date`	The time the report was generated.
`savantID`	Identifier for the Edge Server that generated this report.
`threshold`	The low-cache reporting threshold defined for the `EPCCacheSpec`.

EPCCacheSpecInfo

Java implementation package: `com.connecterra.alepc.api`

Describes administrative information about an EPC cache.

```
subscriberCount : int
pcSpecs : List
activationCount : int
lastActivated : long
```

```
replenishCount : int
lastReplenished : long
lastReported : long
cacheSize : long
cacheContent : EPCPatterns
---
```

Field	Description
`activationCount`	The number of times an EPC value has been obtained from this EPC cache since it was defined.
`cacheContent`	The EPCs in this cache. See EPCPatterns on page 6-12.
`cacheSize`	How many entries remain in the EPC cache.
`lastActivated`	The last time an EPC value was obtained from this EPC cache.
`lastReplenished`	The last time this EPC cache was replenished.
`lastReported`	The last time an `EPCCacheReport` was generated by this EPC cache.
`pcSpecs`	Returns the names of the `PCSpec` instances, if any, that are using this EPC cache.
`replenishCount`	The number of times this EPC cache has been replenished since it was defined.
`subscriberCount`	The number of URIs subscribed to this EPC cache.

EPCPatterns

Java implementation package: `com.connecterra.alepc.api`

A list of EPC pattern URIs.

```
patterns : List
---
```

Field	Description
`patterns`	EPC pattern URIs. The ordering of EPC patterns is significant. See EPC Patterns on page 5-13 and EPC Caches and Pools on page 2-8.

XML Representations

The focal points of the ALE tag writing interface from an application's perspective are the `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` objects. The Edge Server provides a standard way of representing instances of these objects in XML. The XML form of `PCWriteReport` and `EPCCacheReport` is used by most of the asynchronous delivery mechanisms, as described in Chapter 3: Asynchronous Notification Mechanisms. User applications may also find the XML forms very useful as a means of interchange, and for persistent storage.

The XML forms are defined by the XSD schema. This schema is in the RFTagAware installation in the file `share/schemas/ALEPC.xsd`.

The Java binding for ALE provides XML serializer and deserializer classes for translating between the XML representation and the Java representation of the `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` types. Applications may use these facilities to process reports received via the Edge Server's asynchronous delivery mechanisms, and for other purposes. See Using XML Serializers and Deserializers from Java (page 6-16) for more information.

The remainder of this section presents examples of `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` as rendered into XML. These examples include additional line breaks and whitespace for the sake of readability. RFTagAware permits (but does not require) this whitespace when reading XML; usually RFTagAware omits this whitespace when writing XML.

PCSpec - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<PCSpec xmlns="http://schemas.connecterra.com/alepc">

    <!-- The name of the EPC cache from which this PCSpec obtains EPC
         values for tag programming operations. Optional. -->
    <cacheName>mycache</cacheName>

    <!-- Specifies a string to be included in PCWriteReport instances
         generated by this PCSpec.  Optional. -->
    <applicationData>application-specific data here</applicationData>

    <logicalReaders>
        <!-- determines which logical reader(s) will be used by this
             PCSpec.  Logical reader names are defined in edge.props. -->
        <logicalReader>TagWriteStation</logicalReader>
    </logicalReaders>

    <!-- Specifies name/value pairs to be passed down to reader drivers used
         by this PCSpec's programming cycles.  Optional. -->
    <readerParameters>
        <readerParameter name="paramName">paramValue</readerParameter>
        <readerParameter name="anotherParamName">another parameter value</
readerParameter>
    </readerParameters>

    <!-- Determines when this programming cycle starts and stops. -->
    <boundarySpec>
        <!-- Trigger that starts a programming cycle.  Optional -->
        <startTrigger> trigger URI here... </startTrigger>
```

```
        <!-- Trigger that stops a programming cycle.  Optional -->
        <stopTrigger> trigger URI here... </stopTrigger>

        <!-- Specifies maximum number of tag writing trials.  Optional,
             default is unlimited number of trials. -->
        <trials>1</trials>

        <!-- Specifies maximum number of milliseconds to spend retrying failed
             tag writing operations.  Optional, default is no time limit. -->
        <duration>1000</duration>
    </boundarySpec>

</PCSpec>
```

PCWriteReport - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<PCWriteReport date="2004-05-27T18:56:31.179Z"
               savantID="test-edge-server"
               specName="testspec"
               totalMilliseconds="10"
               totalTrials="1"
                xmlns="http://schemas.connecterra.com/alepc">

    <applicationData>application-specific data here</applicationData>

    <wasSuccessful>true</wasSuccessful>

    <status>SUCCESSFUL</status>

    <physicalReaders>
        <physicalReader>tws1</physicalReader>
    </physicalReaders>

    <failedLogicalReaders/>

    <cacheName>mycache</cacheName>

    <cacheSize>11</cacheSize>

    <epc>urn:epc:tag:gid-64-i:1.5.1</epc>

    <successfulLogicalReader>TagWriteStation</successfulLogicalReader>

</PCWriteReport>
```

EPCCacheSpec - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<EPCCacheSpec xmlns="http://schemas.connecterra.com/alepc">

    <!-- Specifies a string to be included in EPCCacheReport instances
         generated by this EPCCacheSpec.  Optional. -->
    <applicationData>cache-specific data here</applicationData>

    <!-- Specifies that when this cache's size drops to (or below) the
```

```
        given number of EPC values, a EPCCacheReport should be issued. -->
    <threshold>2500</threshold>

    <!-- Specifies that EPCCacheReport instances should include the current
        contents of the cache.  Optional, default is false. -->
    <includeCacheContent>true</includeCacheContent>

</EPCCacheSpec>
```

EPCCacheReport - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<EPCCacheReport date="2004-05-27T18:59:32.890Z"
                savantID="test-edge-server"
                xmlns="http://schemas.connecterra.com/alepc">

    <cacheName>mycache</cacheName>

    <applicationData>cache-specific data goes here</applicationData>

    <cacheSize>10</cacheSize>

    <cacheContent>
        <pattern>urn:epc:pat:gid-64-i:1.5.[3-12]</pattern>
    </cacheContent>

    <threshold>2500</threshold>

</EPCCacheReport>
```

XML Schema for PCSpec, PCWriteReport, EPCCacheSpec, and EPCCacheReport

This `share/schemas/ALEPC.xsd` file in the RFTagAware installation defines an XML representation for `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` instances, using the W3C XML Schema language.

Using the ALE Tag Writing API from Java

To use the ALE tag writing API, you create an instance of the `AxisALEPCClient` class provided in the `com.connecterra.alepc.client` package. This class implements the `ALEPC` interface as described in ALEPC: Main Tag Writing Interface on page 6-3 and provides all of the methods described there. It also provides an additional method, `getALEPCFactory`, which returns a factory for creating instances of other types, described below. The `AxisALEPCClient` interacts with an RFTagAware Edge Server over the network using SOAP over HTTP. When you construct an instance of `AxisALEPCClient`, you provide a service URL for the Edge Server with which you wish to interact.

When using the `AxisALEPCClient` class, you need to create instances of `PCSpec`, `EPCCacheSpec`, and other types described in this chapter. The `ALEPCFactory` interface (in package `com.connecterra.alepc.api`) provides methods for creating instances of those types. You obtain an instance of the `ALEPCFactory` interface by calling the `getALEPCFactory` method provided by the `AxisALEPCClient` class. When passing arguments to methods of a specific `AxisALEPCClient` instance, you must always use the factory instance provided by that `AxisALEPCClient` instance.

Using XML Serializers and Deserializers from Java

The Java binding of the ALE API provides some additional utility classes for writing XML representations of the data types used in the ALE API. With these classes, you can convert a Java object representation of a particular data type into XML ("serialization"), and likewise convert an XML representation of a particular data type back into a Java object ("deserialization"). The XML schemas may be found in your RFTagAware installation directory at:

```
/share/schemas/EPCglobal-ale-1_0.xsd
/share/schemas/EPCglobal.xsd
/share/schemas/EPCglobal-ale-1_0-RFTagAware-extensions.xsd
/share/schemas/ALEPC.xsd
```

To read and write XML for types used in the ALE tag writing API, you use an instance of the `PCXMLSerializationFactory` provided in the `com.connecterra.alepc.encoding` package. There is only one static instance of this class, which you obtain using the static method `getInstance()`, with no argument. Using the `PCXMLSerializationFactory` instance, you can create instances of `PCXMLSerializer` and `PCXMLDeserializer` to serialize and deserialize instances of the following classes: `PCSpec`, `PCWriteReport`, `PCSpecInfo`, `PCSubscriptionInfo`, `PCSubscriptionControls`, `EPCCacheSpec`, `EPCCacheReport`, and `EPCCacheSpecInfo`.

Chapter 7: Sample Java Applications

Contents

This chapter describes how to use the sample Java applications provided in your RFTagAware installation. The sample applications illustrate the use of the Java language binding for the ALE interface. Unlike other parts of RFTagAware, the sample applications are free for you to use and modify for your own purposes. You may use them as a starting point for developing your own applications.

Overview

There are several samples provided with RFTagAware:

- `ImmediateSample` — shows how to use the XML serializer and deserializer, and the `ALE immediate` method. The sample program reads an `ECSpec` from an XML file, activates it for one event cycle using the `ALE immediate` method, and displays the results in XML to the console.
- `SubscribeSample` — shows how to use the `ALE subscribe` and `unsubscribe` methods, as well as several other administrative methods within the `ALE` API. The sample provides a simple command-line interface that lets you define `ECSpec` instances from XML files, subscribe a delivery address to a previously defined `ECSpec`, unsubscribe a delivery address, and list existing `ECSpec` instances and subscriptions. As well as illustrating the use of several `ALE` methods, this sample serves as a useful command-line utility program in its own right.
- `ImmediateProgramSample` — shows a simple example of how to use the `ALEPC` API to program an Electronic Product Code (EPC) value into a tag using a specified logical reader. The programming cycle specification is read from an XML file, and the programming cycle reports are printed as XML.
- `ProgrammingSample` — shows how to use the `ALEPC` methods to manipulate Programming Cycles and EPC Caches.

Setting Up Your Development Environment

To compile and run the sample applications, you need to install both the Java Development Kit (JDK™) 1.4 or later, and the RFTagAware software.

For detailed information about system requirements, prerequisite software, and how to install RFTagAware, see the *RFTagAware Deployment Guide*.

Compiling and Running the Samples

The instructions for running all samples are the same:

1. From the `control/bin` subdirectory of your RFTagAware installation, run the following scripts in this order:

   `RunReaderSim` (if you are using the Reader Simulator)

   `RunEdgeServer` (required)

   `RunAdminConsole` (optional)

   These files end with the suffix `.sh` or `.bat`, depending on your platform.

2. Go to the directory for the sample program you want to run (one of the subdirectories within the `samples` subdirectory of your RFTagAware installation).
3. Run the "build" script (`build.sh` or `build.bat` depending on your platform) from the command line. This compiles the sample program.
4. Run the "run" script (`run.sh` or `run.bat` depending on your platform) from the command line. This runs the sample program you just compiled.

   Some samples require additional command line arguments to the "run" script:

   - `SubscribeSample`, see SubscribeSample: Exploring Asynchronous Event Cycle Delivery on page 7-8. The sample program connects to your Edge Server, carries out its task, and then exits.
   - `ImmediateProgramSample`, see ImmediateProgramSample: Writing Tags on page 7-12. You need to provide information about the EPC value you want to write to the tag.
   - `ProgrammingSample`, see ProgrammingSample: Exploring Programming Cycles and EPC Caches on page 7-16. This sample shows you how to manipulate programming cycles and EPC caches.

For some tutorial walk throughs of the samples, see:

ImmediateSample: Getting Started Reading Tags

The `ImmediateSample` program shows how to use the XML serializer and deserializer, and the ALE `immediate` method. The sample program reads an `ECSpec` from an XML file, activates it for one event cycle using the ALE `immediate` method, and displays the results in XML to the console.

In the following description, it is assumed that you are using the Reader Simulator provided with RFTagAware. You may, however, use an actual reader and tags if you have them.

The sample program reads an `ECSpec` from the file `ECSpec.xml` in the sample program folder. The file as provided with RFTagAware is reproduced below. (The file provided with RFTagAware includes comments that are omitted below.) After you become familiar with the sample, you are encouraged to experiment by changing this file to see what happens.

ECSpec.xml example

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

<ale:ECSpec xmlns:ale="urn:epcglobal:ale:xsd:1"
            xmlns:aleext="http://schemas.connecterra.com/EPCglobal-extensions/ale"
            creationDate="2004-11-15T16:18:43.500Z"
            schemaVersion="1.0"
            includeSpecInReports="false" >
<logicalReaders>
   <logicalReader>ConnecTerra1</logicalReader>
</logicalReaders>

<boundarySpec>
   <aleext:durationReadCycles>1</aleext:durationReadCycles>
</boundarySpec>

<reportSpecs>
   <reportSpec reportName="ImmediateSample Report">
   <reportSet set="CURRENT" />
   <output includeCount="true" includeTag="true"/>
   </reportSpec>
</reportSpecs>

<aleext:applicationData>application-specific data here</aleext:applicationData>

</ale:ECSpec>
```

We specify the logical reader as `ConnecTerra1` (which is mapped to "Antenna 1" in the `ReaderSimulator` by default if you installed the Reader Simulator. If you installed your own reader, you may need to change the `ECSpec` to refer to one of your logical readers). We also specify that our event cycle is to be exactly one read cycle — this is far smaller than you are likely to use in any real situation (as we will demonstrate), but we will leave it alone for now to illustrate how the ALE interface works. The final section defines a report specification. Basically, we want to get both a count and a list of all the `CURRENT` tags visible to logical reader `ConnecTerra1`.

Now, run the sample following the instructions in Compiling and Running the Samples on page 7-2. You should see output similar to the following:

```
Immediate Sample, XML-based
sending request to Edge Server...
  ...received response.

Received the following ECReports:

<ale:ECReports ALEID="EdgeServerID" creationDate="2005-01-06T17:01:09.093Z"
date="2005-01-06T17:01:09.093Z" schemaURL="http://schemas.connecterra.com/EPCglobal/
ale-1_0.xsd" schemaVersion="1" specName="$immediate=10"
terminationCondition="DURATION" totalMilliseconds="234"
xmlns:ale="urn:epcglobal:ale:xsd:1" xmlns:aleext="http://schemas.connecterra.com/
EPCglobal-extensions/ale">
 <reports>
  <report reportName="ImmediateSample Report">
   <group>
    <groupList>
```

```
      <member>
       <tag>urn:epc:tag:gid-64-i:10.50.5</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.40.4</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.10.1</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.30.3</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.70.7</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.20.2</tag>
      </member>
      <member>
       <tag>urn:epc:tag:gid-64-i:10.60.6</tag>
      </member>
     </groupList>
     <groupCount>
      <count>7</count>
     </groupCount>
    </group>
   </report>
  </reports>
  <aleext:applicationData>application-specific data here</aleext:applicationData>
  <aleext:failedLogicalReaders/>
  <aleext:physicalReaders>
   <aleext:physicalReader>SimReadr</aleext:physicalReader>
  </aleext:physicalReaders>
  <aleext:totalReadCycles>1</aleext:totalReadCycles>
</ale:ECReports>Press any key to continue . . .
```

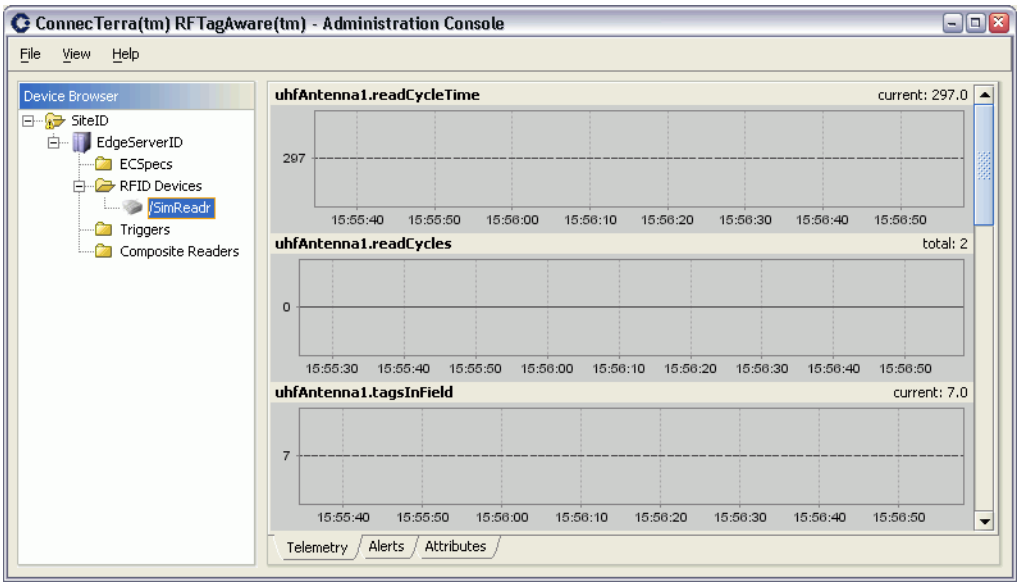
The number of `epc` elements in the list report should be equal to the number of tags you have checked under "Antenna 1" in the Reader Simulator. (If you are using a real reader, you may not see all the tags you have placed near your antenna.)

Using ImmediateSample With the Administration Console

If you are running the Administration Console, you may want to run `ImmediateSample` again, watching its effects as described below:

1. First, set up your desktop so you can see both the Administration Console and the `ImmediateSample` console window at the same time.
2. In the Administration Console, click **SimReadr** in the device browser on the left, then click the **Telemetry** tab in the right pane.

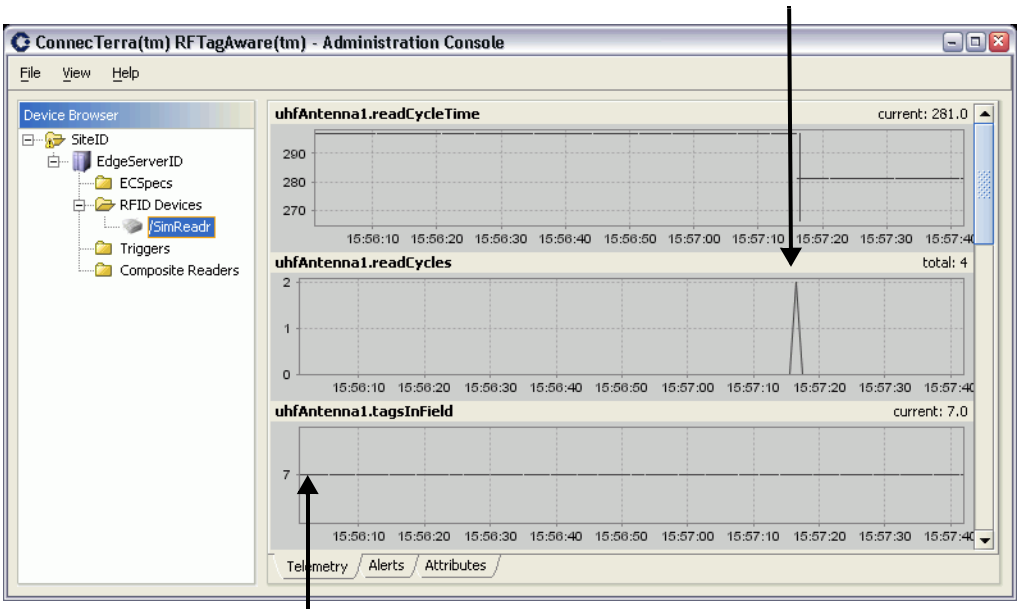
Reader Telemetry Tab



Keep your eye on the *uhfAntenna1.readCycles* display. uhfAntenna1 corresponds to the logical reader *ConnecTerra1* that we specified in the sample `ECSpec.xml`. Looking at this display will show you when the `ImmediateSample` program activates the antenna for one event cycle, using the ALE `immediate` method

3. Run the sample using the instructions in Compiling and Running the Samples on page 7-2.

ImmediateSample's event cycle was set to just one read cycle,shown below.



uhfAntenna1 is seeing the seven tags that the Reader Simulator is configured for.

You will see that uhfAntenna1 (logical reader ConnecTerra1) is activated for exactly one read cycle — which is exactly what we specified for a `boundarySpec` in `ECSpec.xml`:

```
<boundarySpec>
        <durationReadCycles>1</durationReadCycles>
</boundarySpec>
```

ImmediateSample: Event Cycles and Reliability

We will now use the `ImmediateSample` application to illustrate some aspects of event cycles and how they can be used to address situations where not every tag can be read in a single read cycle. This is a very common situation, and can arise either because of the inherently unreliable nature of RFID tags, or because the business situation simply implies that not all tags for an application level event are in front of the antenna at the same time (for example, because a large pallet is moving slowly past an antenna).

To simulate this situation, we will use the "reliability" field provided as part of the Reader Simulator. Change the Reliability field in the Reader Simulator to 50%. This tells the Reader Simulator to report each selected tag with only 50% probability in any given read cycle. Now run ImmediateSample as you did in ImmediateSample: Getting Started Reading Tags on page 7-3. In all likelihood, you will see fewer tags in the report than you did previously.

Now, we will see how the event cycle combines tags from multiple read cycles into a single report, and how this counteracts the limitations of dealing with read cycles individually. Follow these steps:

1. Open the file `ECSpec.xml` in a text editor.
2. Change the line that reads:
   `<aleext:durationReadCycles>1</aleext:durationReadCycles>`

   so that it reads:
   `<aleext:durationReadCycles>3</aleext:durationReadCycles>`
3. Save the file.
4. Leave the reliability on the Reader Simulator set to 50%.

Now run the sample again. This time, you should see most if not all of the tags.

It is usually difficult to guess how many read cycles are required to read all tags of interest. In some cases, external events dictate which read cycles should be grouped into an event cycle — the `startTrigger` and `stopTrigger` features of the ALE interface (see ECBoundarySpec on page 5-7) may be used for this purpose. In other cases, you want an event cycle to continue as long as needed until all tags have been read. In such cases, you can use the `stableSetInterval` feature of the ALE interface.

ImmediateSample: Reading from Different Readers

The ALE interface makes it very easy to select different readers without altering application code, even changing the number of readers. To illustrate, follow these steps:

1. Open the file `ECSpec.xml` in a text editor.
2. Immediately after the line that reads:
   `<logicalReaderName>ConnecTerra1</logicalReaderName>`

   add a second line so that together they look like this:
   ```
   <logicalReaderName>ConnecTerra1</logicalReaderName>
   <logicalReaderName>ConnecTerra2</logicalReaderName>
   ```
3. Save the file.

Now run `ImmediateSample` again. In the report, you'll see tags read from both readers.

SubscribeSample: Exploring Asynchronous Event Cycle Delivery

The `SubscribeSample` program shows how to use the ALE `subscribe` and `unsubscribe` methods, as well as several other administrative methods within the ALE API. The sample provides a simple command line interface that lets you define `ECSpec` instances from XML files, subscribe a delivery address to a previously defined `ECSpec`, unsubscribe a delivery address, and list existing `ECSpec` instances and subscriptions. As well as illustrating the use of several ALE methods, this sample serves as a useful command line utility program in its own right.

Like the `ImmediateSample`, the `SubscribeSample` works with XML files to define event cycle specifications. However, `SubscribeSample` differs from `ImmediateSample` in several respects:

- Any number of event cycle specifications can be defined, each with their own name. You invoke the `SubscribeSample` program with the `define` command for each event cycle you want to define.
- To obtain event cycle reports, you add one or more subscribers for the event cycle(s) you have defined, by invoking the `SubscribeSample` program with the `subscribe` command.
- Once you have defined event cycle(s) and added one or more subscriptions, the Edge Server executes event cycles and sends reports to the subscribers. This takes place asynchronously, even when the `SubscribeSample` program is not running.

Here are step-by-step instructions for working with the `SubscribeSample` program.

1. From the `control/bin` subdirectory of your RFTagAware installation, run the following scripts in this order:

   `RunReaderSim` (if you are using the Reader Simulator)

   `RunEdgeServer` (required)

   `RunAdminConsole` (optional, but strongly suggested for this tutorial)

   These files end with the suffix `.sh` or `.bat`, depending on your platform.

2. Find the console window for the Edge Server and leave it open on your desktop. Later you will be looking at console subscriber output sent to this window.

3. Go to the RFTagAware directory:
   `samples/SubscribeSample`

4. In a shell, type:
   `./run.sh define mycmdlinespec ECSpec.xml`

   (On Windows, type `.\run.bat` instead of `./run.sh`. Do this replacement for the rest of the examples in this section.)

   You will see some output messages from the `SubscribeSample` program indicating that an event cycle specification has been defined. At this point, the `ECSpec` is defined but is not active, because there are no subscribers.

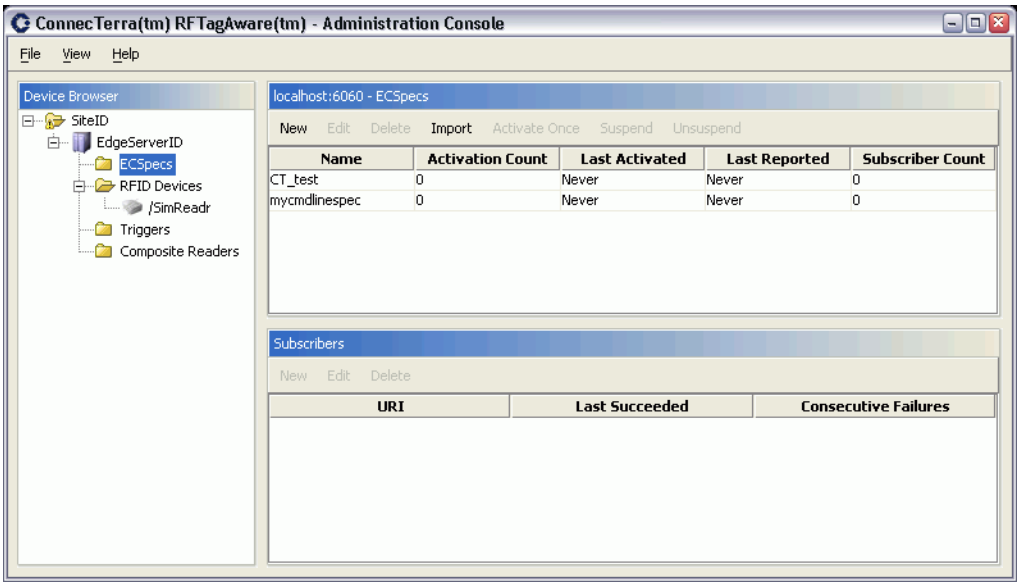
   Note: You can define as many different `ECSpec` instances as you want, as long as you give them distinct names. We used the name `mycmdlinespec` for the `ECSpec` we defined; we will be referring to this `ECSpec` again using its name.

5. If you are running the Administration Console, set up your desktop so you can see both the Administration Console and the `SubscribeSample` shell at same time.

   In the Administration Console, click **ECSpecs** in the device browser. Note that the `ECSpec` you just defined (`mycmdlinespec`) is listed in the right pane.

   Note: Defining an `ECSpec` is NOT the same as activating it. You have not yet told a reader to read any tags, or done anything else with the `ECSpec` yet. You have simply defined a set of actions (read cycles, delivery activities, and so on) that can take place some time in the future once the `ECSpec` is activated by a method such as `immediate`, `poll`, or, in this example, `subscribe`.

ECSpecs Pane



6. To prove that defining and activating an ECSpec are different, display the telemetry tab for the Reader Simulator, and then define a second `ECSpec` in the `SubscribeSample` shell:
   `./run.sh define myspec2 ECSpec.xml`

   Keep your eye on the *uhfAntenna1.readCycles* telemetry trace. You will not see any read cycles take place. (This *uhfAntenna1.readCycles* trace corresponds to the logical reader that `ECSpec.xml` is referencing.)

7. To demonstrate some other features of `SubscribeSample`, return to the `SubscribeSample` shell and type:
   `./run.sh list-specs`

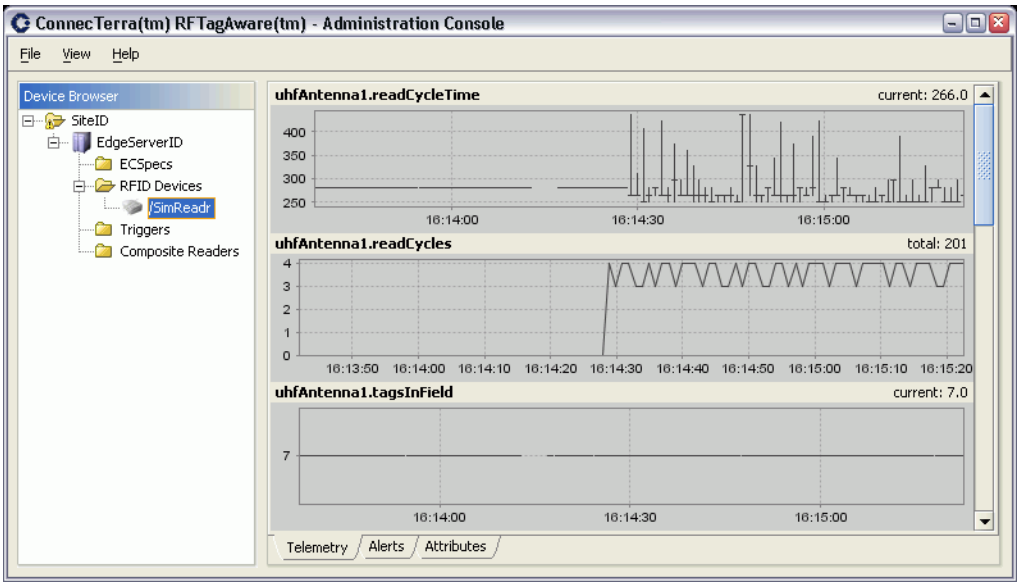
   This prints a list of the names of the `ECSpec` instances that are currently defined in the Edge Server. You should see `mycmdlinespec` and any other event cycle specifications you have defined.

8. In the shell, type:
   `./run.sh subscribe mycmdlinespec console:test`

   Look in the Edge Server window — the Edge Server is now printing event cycle reports to the console.

   Also, take a look at the Administration Console telemetry display:

Reader Simulator Telemetry (activated ECSpec



As you can see, the `subscribe` method that you invoked when you ran `SubscribeSample` this last time has activated the Reader Simulator, and it is now performing read cycles as specified in the `ECSpec` called `mycmdlinespec`.

9. Now we'll experiment with a different kind of event delivery driver. Create a new directory in a file system accessible to the Edge Server. For example:
   ```
   mkdir /tmp/ale
   ```
   On the Windows platform, the equivalent command would be:
   ```
   mkdir c:\temp\ale
   ```

10. In the shell, type:
    ```
    ./run.sh subscribe mycmdlinespec file:///tmp/ale
    ```
    or on the Windows platform, type:
    ```
    .\run.bat subscribe mycmdlinespec file:///c:/temp/ale
    ```

11. Use a file system tool to examine the contents of the `/tmp/ale` (or `c:\temp\ale`) directory. You will see that the Edge Server is creating XML files, each containing a single event cycle report. Alternately, if the subscription URI were to refer to a file (as opposed to a directory), then the successive event cycle reports would be appended to that file.

12. In the shell, type:
    ```
    ./run.sh list-subs mycmdlinespec
    ```
    This prints a list of the URIs that have been subscribed to the `ECSpec` named `mycmdlinespec`.

13. In the shell, type:
    ```
    ./run.sh unsubscribe mycmdlinespec console:test
    ```
    Look in the Edge Server window — the Edge Server is no longer printing event cycle reports to its console. But look in the temporary directory you created earlier — the Edge Server is still writing XML report files into this directory, because the other subscription is still active.

SubscribeSample Command Line Options

For information about `SubscribeSample`'s command line options, you can navigate to the `SubscribeSample` directory and type `run`. This displays the command help shown below. Note that the help distinguishes EPCglobal functions from RFTagAware extensions.

```
Usage:
EPCglobal ALE 1.0 commands
define <specName> <ecSpecFilename>
or   undefine <specName>
or   getECSpec <specName>
or   getECSpecNames
or   subscribe <specName> <notificationURI>
or   unsubscribe <specName> <notificationURI>
or   getSubscribers <specName>
or   poll <specName>
or   immediate <ecSpecFilename>
or   getStandardVersion
or   getVendorVersion

RFTagAware extensions:
get-spec-info <specName>
or   redefine <specName> <ecSpecFilename>
or   suspend <specName>
or   unsuspend <specName>
or   stop <specName>
```

ImmediateProgramSample: Writing Tags

This sample shows how to use the ALE API to program an Electronic Product Code (EPC) value into a tag using a specified logical reader. The programming cycle specification is read from an XML file, and the programming cycle reports are printed as XML. You can run this sample with the simulator, or with any of the printers or readers for which RFTagAware supports tag writing. See the supported RFID readers section of the *RFTagAware Reader Configuration Guide* for this information.

If you plan to run this sample with the simulator, see Using ImmediateProgramSample with the Reader Simulator on page 7-15.

Here are step-by-step instructions for working with the `ImmediateProgramSample` program.

1. From the `control/bin` subdirectory of your RFTagAware installation, run the following scripts in this order:

   `RunReaderSim` (if you are using the simulator)

   `RunEdgeServer` (required)

   `RunAdminConsole` (optional)

   These files end with the suffix `.sh` or `.bat`, depending on your platform.

If you are using the simulator, be sure to read the section Using ImmediateProgramSample with the Reader Simulator on page 7-15.

2. Find the console window for the Edge Server and leave it open on your desktop. Later you will be looking at output that this sample program sends to this window.

3. Go to the RFTagAware directory:
   `samples/ImmediateProgramSample`

   This sample uses the file `PCSpec.xml` as part of its input. This file defines the programming cycle (see PCSpec on page 6-5). Part of the file is reproduced here — you can take a look at the complete file in the `samples` directory:

PCSpec.xml example

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

<PCSpec xmlns="http://schemas.connecterra.com/alepc">

    <!-- Specifies a string to be included in PCWriteReport instances
         generated by this PCSpec.  Optional. -->
    <applicationData>application specific data can go here</applicationData>

    <logicalReaders>
        <!-- determines which logical reader(s) will be used by this
             programming cycle.  Logical reader names are defined in edge.props. -->
        <logicalReader>TagWriteStation</logicalReader>
    </logicalReaders>

    <boundarySpec>
        <!-- the boundarySpec determines when this programming cycle starts
             and stops.  Because this sample program uses
             ALEPC.immediate(), a programming cycle starts whenever
             ALEPC.immediate() is called. -->

        <!-- Specifies maximum number of tag writing trials.  Optional,
             default is unlimited number of trials. -->
        <trials>1</trials>

        <!-- Specifies maximum number of milliseconds to spend retrying failed
             tag writing operations.  Optional, default is no time limit. -->
        <duration>4000</duration>
    </boundarySpec>

</PCSpec>
```

4. With a text editor, edit `PCSpec.xml` to replace the "placeholder" line:
   `<logicalReader>TagWriteStation</logicalReader>`

   with the logical reader you will actually use to write the tag. For example, if you are using the Reader Simulator, you might specify:
   `<logicalReader>ConnecTerra1</logicalReader>`

5. In a shell, type:
   `./run.sh epcValue`

   where `epcValue` is the EPC to write to the tag (e.g., `urn:epc:tag:gid-64-i:1.4.10`).

   (On Windows, type `.\run.bat` instead of `./run.sh`. Do this replacement for the rest of the examples in this section.)

6. In the console window, you should see output similar to the following:

ImmediateProgramSample Run Output

```
C:\Program Files\ConnecTerra\RFTagAware\1.2\samples\ImmediateProgramSample>run
urn:epc:tag:gid-64-i:1.4.10
Immediate Program Sample, XML-based
  sending request to ALE engine...
  ...received response.

Received the following PCWriteReport:

<?xml version="1.0" encoding="UTF-8"?>
<PCWriteReport date="2004-05-27T19:14:34.597Z" savantID="EdgeServerID"
specName="$immediate=3" totalMilliseconds="10" to
talTrials="1" xmlns="http://schemas.connecterra.com/alepc">
 <applicationData>application specific data can go here</applicationData>
 <wasSuccessful>true</wasSuccessful>
 <status>SUCCESSFUL</status>
 <physicalReaders>
  <physicalReader>alr1</physicalReader>
 </physicalReaders>
 <failedLogicalReaders/>
 <cacheSize>0</cacheSize>
 <epc>urn:epc:tag:gid-64-i:1.4.10</epc>
 <successfulLogicalReader>Alien1</successfulLogicalReader>
</PCWriteReport>Press any key to continue . . .
```

The console output includes a `PCWriteReport`, expressed in XML. (See PCWriteReport on page 6-7.) `PCWriteReport` describes the programming cycle's tag writing operation.

First, the `<applicationData>` element displays the information that the originating `PCSpec.xml` included in its `<applicationData>` element.

In this example, the `<wasSuccessful>` element (set to `true`) indicates that this programming cycle was successful. The `<status>` element is correspondingly set to `SUCCESSFUL`.

If the programming cycle had encountered problems, the `<status>` element would have provided diagnostic information about the termination status of programming cycle (`CACHE_EMPTY`, `READER_ERROR`, and so on. (See PCStatus on page 6-9.)

The `<physicalReaders>` element indicates which physical readers were involved in this tag writing operation, in this case just one physical reader, `alr1`.

`<failedLogicalReaders>` is empty, because no logical readers failed during this programming cycle.

`<cacheSize>` is set to zero — in this simple example, you passed in an EPC value as a parameter to the sample program, the programming cycle used this value, and there are no other values available. In other situations `<cacheSize>` will tell you how many EPC values are left in the EPC cache associated with the originating `PCSpec`. (See EPCCacheSpec on page 6-10.)

`<epc>` displays the EPC value that was written to the tag, in this case:
`urn:epc:tag:gid-64-i:1.4.10`

Finally, `<successfulLogicalReader>` indicates that the logical reader `Alien1` was the logical reader that wrote this tag.

Using ImmediateProgramSample with the Reader Simulator

You can use the Reader Simulator to simulate tag writes with the `ImmediateProgramSample` application. This section contains notes on how to do this.

The `PCSpec.xml` files in the `samples/ImmediateProgramSample` and `samples/ProgrammingSample` directories use `TagWriteStation` as the logical reader name on which the tags will be written. This logical reader is not defined during the product installation. There are a number of ways to address this:

- Edit the `PCSpec.xml` file in the appropriate samples directory to change the name of the `logicalReader` to the name of your Reader Simulator, or
- Edit the `edge.props` file to add/change the name of a simulated reader to `TagWriteStation`, or
- Edit the `edge.props` file to add a composite reader called `TagWriteStation`, where the only member of this composite reader is one antenna of your simulated reader. For example, using the default install, you would add the line
  `com.connecterra.ale.compositeReader.TagWriteStation.members=ConnecTerra1`

The Reader Simulator provides support for writing six tag types only: GID-64-i, GID-96, SGTIN-64, SGTIN-96, SSCC-64, and SSCC-96. The Reader Simulator needs access to a valid Company Prefix Index Table to process SGTIN-64 and SSCC-64 tags. This file can be specified in the `bin/RunReaderSim` (`.bat`/`.sh`) script as one of the command parameters to the Java invocation:

```
-epcIndexTableURL http://onsepc.com/ManagerTranslation.xml
```

This file must be the same as the value of the `com.connecterra.ale.epcIndexTable` property in the `etc/edge.props` file. If the two files are different, then unpredictable results may occur.

As with a real RFID reader, there must be only one tag to be written in range of the antenna. With the Reader Simulator, you must unselect all but one of the tags checked in the GUI. Otherwise the tag write will fail with a `MULTIPLE_IN_FIELD` error.

GID-64-i tags are outside the *EPCglobal Tag Data Standards*. For standard tags, there are strict definitions of what are valid data in the various fields of the tag. This is one area where leading zeros are considered important. The following non-normative descriptions are provided for guidance — the document referenced above is definitive.

An SGTIN-64 tag is made up of a Filter field, a Company Prefix, an Item Reference code and a Serial Number. The Company Prefix and the Item Reference together must total 13 decimal digits. So this is a valid tag:

```
urn:epc:tag:sgtin-64:1.5413149.000001.1
```

while this is an invalid tag:

```
urn:epc:tag:sgtin-64:1.5413149.1.1
```

An SSCC-64 tag is made up of a Filter field, a Company Prefix and a Serial Reference. The Company Prefix and the Serial Reference together must total 17 decimal digits. So this is a valid tag:

```
urn:epc:tag:sscc-64:1.0353265.0000010000
```

while this is an invalid tag:

```
urn:epc:tag:sscc-64:1.0353265.100000
```

When using the sample code, any attempt to write a poorly formatted tag may generate a non-specific `java.net.URISyntax` exception with the (example) detail:

```
non valid uri syntax for epc tag: null: urn:epc:tag:sscc-64:1.0353265.100000
```

The Reader Simulator does not support other 64-bit or 96-bit tag types for writing. Any attempt to write (for example) a GRAI-64, GIAI-64, or SGLN-64 tag will fail.

ProgrammingSample: Exploring Programming Cycles and EPC Caches

The `ProgrammingSample` program shows how to use the ALEPC methods to manipulate Programming Cycles and EPC Caches. The sample provides a simple command line interface that lets you define `PCSpec` instances from XML files, subscribe or unsubscribe a delivery address to a previously defined `PCSpec` to receive `PCWriteReport` instances, and list existing `PCSpec` instances and subscriptions. In addition, you can define `EPCCacheSpec` instances from XML files, subscribe or unsubscribe for `EPCCacheReport` instances, and replenish or deplete defined EPC caches.

As well as illustrating the use of several ALEPC methods, this sample serves as a useful command line utility program in its own right.

Like the `ImmediateProgramSample`, the `ProgrammingSample` works with XML files to define programming cycle specifications. However, `ProgrammingSample` differs from `ImmediateProgramSample` in several respects:

- The `ProgrammingSample` uses EPC caches to obtain EPC values to be programmed to tags. You invoke the `ProgrammingSample` program with the `define-cache` command for each EPC cache you want to define, and use the `replenish` command to load an EPC cache with EPC patterns that define its contents.
- Any number of programming cycle specifications can be defined, each with its own name. You invoke the `ProgrammingSample` program with the `define` command for each programming cycle you want to define.
- To obtain programming cycle write reports, you add one or more subscribers for the programming cycle(s) you have defined, by invoking the `ProgrammingSample` program with the `subscribe` command.
- To obtain cache-low reports, you add one or more subscribers for the EPC cache(s) you have defined, by invoking the `ProgrammingSample` program with the `subscribe-cache` command.
- Once you have defined programming cycle(s), EPC cache(s), and added one or more subscriptions, you invoke the `ProgrammingSample` program with the `poll` command to cause a programming cycle to commence. The `PCSpec` you poll will obtain an EPC value from its associated EPC cache and perform a tag programming operation using that EPC value.

Here are step-by-step instructions for working with the `ProgrammingSample` program.

1. Configure the Edge Server to use the Reader Simulator or any of the printers or readers for which RFTagAware supports tag writing. See the supported RFID readers section of the *RFTagAware Reader Configuration Guide* for information.

   Assign this reader the logical reader name `TagWriteStation`, which is the logical reader name specified in the `ProgrammingSample`'s `PCSpec.xml` file that we will use later. Alternately, you can pick a different logical reader name, as long as you edit `edge.props` and `PCSpec.xml` to both reflect the logical reader name you chose. If you are using the Reader Simulator, please read the section Using ImmediateProgramSample with the Reader Simulator on page 7-15 to understand the constraints of the simulator.

2. From the control/bin subdirectory of your RFTagAware installation, run the following scripts in this order:

   `RunReaderSim` (if you are using the simulator)

   `RunEdgeServer` (required)

   `RunAdminConsole` (optional)

   These files end with the suffix `.sh` or `.bat`, depending on your platform.

3. Find the console window for the Edge Server and leave it open on your desktop. Later you will be looking at console subscriber output sent to this window.

4. Go to the RFTagAware directory:
   `samples/ProgrammingSample`

5. In a shell, type:
   `./run.sh define-cache mycache CacheSpec.xml`

(On Windows, type `.\run.bat` instead of `./run.sh`. Do this replacement for the rest of the examples in this section.)

You will see an output message from the `ProgrammingSample` indicating that an EPC cache has been defined.

Note: You can define as many different EPC caches as you want, as long as you give them distinct names. In the command line above, we gave the name `mycache` for the EPC cache we defined; we will shortly be defining a `PCSpec` that refers to this cache.

6. In a shell, type:
```
./run.sh list-caches
```
This prints a list of the names of the EPC caches that are currently defined in the Edge Server. You should see `mycache` and any other EPC caches you have defined.

7. In a shell, type:
```
./run.sh subscribe-cache mycache console:test
```
Look in the Edge Server window — a low-cache report has been issued to the subscription we just created:

```
<!-- test -->
<?xml version="1.0" encoding="UTF-8"?>
<EPCCacheReport date="2004-06-10T14:04:03.040Z" savantID="EdgeServerID"
xmlns="http://schemas.connecterra.com/alepc">
 <cacheName>mycache</cacheName>
 <applicationData>application-specific data here</applicationData>
 <cacheSize>0</cacheSize>
 <cacheContent/>
 <threshold>10</threshold>
</EPCCacheReport>
```

A low-cache report was issued because the cache we defined does not yet have any EPCs, and so is below the low-cache reporting threshold (10) defined in `CacheSpec.xml`. Whenever a cache is below its reporting threshold, it issues low-cache reports to its subscribers. In this case, such a report was issued as soon as a new subscriber was defined.

8. In a shell, type:
```
./run.sh replenish mycache urn:epc:pat:gid-64-i:1.5.[1-15]
```
This stocks the EPC cache we defined earlier with a range of 15 EPC values.

9. In a shell, type:
```
./run.sh cache-info mycache
```

The ProgrammingSample prints:

```
info for EPC cache mycache: Received the following EPCCacheSpecInfo:
<?xml version="1.0" encoding="UTF-8"?>
<EPCCacheSpecInfo xmlns="http://schemas.connecterra.com/alepc">
 <subscriberCount>1</subscriberCount>
 <pcSpecs/>
 <activationCount>0</activationCount>
 <replenishCount>1</replenishCount>
 <lastReplenished>2004-06-10T14:28:34.495Z</lastReplenished>
 <lastReported>2004-06-10T14:04:03.040Z</lastReported>
 <cacheSize>15</cacheSize>
 <cacheContent>
  <pattern>urn:epc:pat:gid-64-i:1.5.[1-15]</pattern>
 </cacheContent>
</EPCCacheSpecInfo>
```

We can see that the EPC cache we defined has one subscriber, no `PCSpec` instances using it (yet), has been replenished once but never activated (used to write tags), and is currently stocked with 15 EPCs from a single range pattern.

10. In a shell, type:
    ```
    ./run.sh define myspec PCSpec.xml
    ```
    You will see an output message from the `ProgrammingSample` indicating that a `PCSpec` has been defined.

    Note: You can define as many different `PCSpec` instances as you want, as long as you give them distinct names. In the command line above, we gave the name `myspec` for the `PCSpec` we defined.

11. In a shell, type:
    ```
    ./run.sh cache-info mycache
    ```
    The `ProgrammingSample` prints information about `mycache`, similar to what was printed earlier. The important difference is that the empty `<pcSpecs/>` element has been replaced with:
    ```
    <pcSpecs>
      <pcSpec>myspec</pcSpec>
    </pcSpecs>
    ```
    This indicates that the `PCSpec` we just defined is using the `mycache` EPC cache we defined earlier. Whenever `myspec` performs a tag programming operation, it will obtain an EPC value from `mycache`.

12. Place a single writable RFID tag in the field of the RFID reader you configured the Edge Server to use.

13. In a shell, type:
    ```
    ./run.sh poll myspec
    ```
    The ProgrammingSample performs a tag programming operation and, if successful, prints a `PCWriteReport` similar to:

    ```
    polling myspec...
      ...received response.
    ```

```
Received the following PCWriteReport:

<?xml version="1.0" encoding="UTF-8"?>
<PCWriteReport date="2004-06-10T14:43:31.525Z" savantID="EdgeServerID"
specName="myspec" totalMilliseconds="10" totalTrials="1" xmlns="http://
schemas.connecterra.com/alepc">
 <applicationData>application-specific data here</applicationData>
 <wasSuccessful>true</wasSuccessful>
 <status>SUCCESSFUL</status>
 <physicalReaders>
  <physicalReader>console</physicalReader>
 </physicalReaders>
 <failedLogicalReaders/>
 <cacheName>mycache</cacheName>
 <cacheSize>14</cacheSize>
 <epc>urn:epc:tag:gid-64-i:1.5.1</epc>
 <successfulLogicalReader>TagWriteStation</successfulLogicalReader>
</PCWriteReport>
```

The report indicates the status of the tag programming operation, and if successful (as in the example above), contains the EPC value that was written to the tag, and also indicates how many EPC values remain in the EPC cache. In this example, note that the EPC cache, which previously had 15 EPC values, now has only 14 EPC values remaining.

14. Repeat the `poll` command several more times, and watch the Edge Server's console window. At some point, the number of EPC values remaining in the cache will drop to the reporting threshold (10), and a low-cache report will be issued to the console subscriber you defined earlier. Each subsequent poll operation will cause a further low-cache report to be issued, unless you first use the replenish command to re-stock the EPC cache.

15. Keep repeating the poll command until the EPC cache is empty, as indicated in the `PCWriteReport` indicating a tag programming failure:

```
polling myspec...
  ...received response.

Received the following PCWriteReport:

<?xml version="1.0" encoding="UTF-8"?>
<PCWriteReport date="2004-06-10T14:59:03.385Z" savantID="EdgeServerID"
specName="myspec" totalMilliseconds="6289" totalTrials="1" xmlns="http://
schemas.connecterra.com/alepc">
 <applicationData>application-specific data here</applicationData>
 <wasSuccessful>false</wasSuccessful>
 <status>CACHE_EMPTY</status>
 <physicalReaders>
  <physicalReader>console</physicalReader>
 </physicalReaders>
 <failedLogicalReaders/>
 <cacheName>mycache</cacheName>
 <cacheSize>0</cacheSize>
 <failureInfo>EPC cache 'mycache' empty for PCSpec myspec</failureInfo>
 <terminationCondition>FAILURE</terminationCondition>
</PCWriteReport>
```

JMS Samples

The samples in this section show how to configure JMS options and naming properties on the RFTagAware Edge Server for vendor-specific JNDI (Java Naming and Directory Interface) providers and JMS servers. The samples also provide deployment units to be deployed into J2EE application servers in enterprise systems and provides sample message receiving programs for message queue servers.

For vendor-specific J2EE application servers, a `JMSTest.ear` enterprise archive file is available for deployment. The enterprise archive file contains a Message Driven Bean (`JMSTestMDB`) which receives JMS messages from specified queues and prints them out to the console.

For message queue servers (WebSphere MQ and TIBCO Enterprise for JMS), sample message receiver programs are provided to receive JMS messages from specified queues and to print them out to the console.

BEA

WebLogic Server

To run the sample JMSTest enterprise program within a BEA WebLogic Application Server deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/BEA/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables.
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify the `edge.props` file in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for `naming.props`.
   - (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
`wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar`

You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
`wrapper.java.classpath.31=../AppServerLib/someJarFile.jar`

Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.

3. Build the sample JMSTest enterprise archive by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).
4. Deploy the sample JMSTest enterprise archive into the Application Server:
   - Start the application server via the `startWebLogic` command.
   - Copy `JMSTest.ear` from `/samples/JMSSamples/BEA/deploy` into the `user_projects/domains/mydomain/applications` directory of the BEA WebLogic installation directory.
   - Using the BEA WebLogic Server Administration Console (`http://<wl_host>:7001/console`), install the `JMSTest.ear` as follows:
     `mydomain, Deployments, Applications, Deploy a New Application, applications`

     Select and deploy `JMSTest.ear`
5. Run the Edge Server.
6. Define an `ECSpec` to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec` (via `run define myECSpec ECSpec.xml`)
7. Set a JMS subscriber to the defined `ECSpec`.

   For example, set a JMS subscriber for myECSpec reports (via `run subscribe myECSpec jms:/queue/weblogic.jms.ConnectionFactory/jms%2FTestQ`)

   Note: BEA provides `weblogic.jms.ConnectionFactory` and `weblogic.jms.XAConnectionFactory` as default connection factories.
8. View `JMSTest MDB` messages showing `ECReports` for the defined `ECSpec` in the console corresponding to the startWebLogic command.

IBM

Application Server

To run the sample JMSTest enterprise program within an IBM WebSphere Application Server deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/IBM/<was_configuration>/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify `edge.props` in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for `naming.props`.
   - (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

     For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar
     ```

     You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=../AppServerLib/someJarFile.jar
     ```

     Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.
   - Modify `RunEdgeServer.bat` or `RunEdgeServer.sh` in the `control/bin` directory of the RFTagAware installation directory to set the `JAVA_HOME` environment variable to the fully-qualified path of an IBM JRE.

     Note that if an IBM JRE is not specified, `CORBA.INITIALIZE` errors will be reported.
3. Build the sample `JMSTest` enterprise archive by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).

4. Deploy the sample `JMSTest` enterprise archive into the Application Server:
   - Start the application server.
   - Using the WebSphere Application Server Administration Console (`http://<was_host>:9090/admin`), install `JMSTest.ear` from `/samples/JMSSamples/IBM/<was_configuration>/deploy` directory as follows:

     Applications, Install New Application, Local Path:

     Specify fully-qualified path for `JMSTest.ear`

     Next, Next, Check "Deploy EJBs"

     Next, Next, Specify Listener Port Name (`JMSTestListener`)

     Next, Next, Finish
5. Run the Edge Server.
6. Define an `ECSpec` to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec` (via `run define myECSpec ECSpec.xml`)
7. Set a JMS subscriber to the defined `ECSpec`.

   For example, set a JMS subscriber for `myECSpec` reports (via `run subscribe myECSpec jms:/queue/jms%2FTestQCF/jms%2FTestQ`)
8. View `JMSTest` MDB messages showing `ECReports` for the defined `ECSpec` in the `logs/server1/SystemOut.log` file of the WebSphere Application Server installation directory.

WebSphere MQ

To run the MQReceiver sample program within an IBM WebSphere MQ deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/IBM/Standalone_WMQ_Sample/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables.
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.

- Modify the `edge.props` file in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for `naming.props`.
- (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

  For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
  `wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar`

  You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
  `wrapper.java.classpath.31=../AppServerLib/someJarFile.jar`

  Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.
- Modify `RunEdgeServer.bat` or `RunEdgeServer.sh` in the `1.2/bin` directory of the RFTagAware installation directory to set the `JAVA_HOME` environment variable to the fully-qualified path of an IBM JRE.

Note: If an IBM JRE is not specified, `CORBA.INITIALIZE` errors will be reported.

3. Build the `MQReceiver` sample program by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).
4. Run `/samples/JMSSamples/IBM/Standalone_WMQ_Sample/MQReceiver.bat` (or `.sh`) to start a queue receiver for a specified queue name (for example, `MQReceiver MQ_JMS_Q`).
5. Run the Edge Server.
6. Define an ECSpec to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec`
   (via `run define myECSpec ECSpec.xml`)
7. Set a JMS subscriber to the defined ECSpec.
   - For example, set a JMS subscriber for myECSpec reports (via `run subscribe myECSpec jms:/queue/jms%2FTestQCF/jms%2FTestQ`)
   - For an LDAP provider, the JMS URI would take the following form:
     `jms:/queue/cn=MQTestQCF/cn=MQTestQ`
8. View `MQReceiver` messages showing `ECReports` for the defined `ECSpec` in the console where `MQReceiver` was started.

JBoss

Application Server

To run the sample JMSTest enterprise program within a JBoss Application Server deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/JBoss/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify the `edge.props` file in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for `naming.props`.
   - (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

     For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar
     ```

     You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=../AppServerLib/someJarFile.jar
     ```

     Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.
3. Build the sample `JMSTest` enterprise archive by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).

4. Deploy the sample `JMSTest` enterprise archive into the Application Server:
   - Run `build.bat` in the `/samples/JMSSamples/JBOSS` subdirectory.
   - Copy `JMSTest.ear` from `/samples/JMSSamples/JBOSS/deploy` into `server/default/deploy` of the JBoss installation directory.
   - Start the application server to automatically deploy the `JMSTest` enterprise archive.
5. Run the Edge Server.
6. Define an ECSpec to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec` (via `run define myECSpec ECSpec.xml`)
7. Set a JMS subscriber to the defined ECSpec.

   For example, set a JMS subscriber for `myECSpec` reports (via `run subscribe myECSpec jms:/queue/ConnectionFactory/queue%2FTestQ`)
8. View `JMSTest` MDB messages showing `ECReports` for the defined `ECSpec` in the console where the JBoss Application Server was started.

Sun

Java System Application Server

To run the sample JMSTest enterprise program within a Sun Java System Application Server deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/Sun/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables.
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify the `edge.props` file in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for naming.props.

- (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

  For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
  `wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar`

  You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
  `wrapper.java.classpath.31=../AppServerLib/someJarFile.jar`

  Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.

3. Build the sample `JMSTest` enterprise archive by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).
4. Deploy the sample `JMSTest` enterprise archive into the Application Server:
   - Start the application server (default server).
   - Using the Sun Application Server Administration Console (`http://<sun_host>:4848/asadmin`), install `JMSTest.ear` from `/samples/JMSSamples/Sun/deploy` directory as follows:

     Application Server, Enterprise Applications, Deploy:

     Specify fully-qualified path for `JMSTest.ear`

     Next

     Specify Application Name, enabled status

     Finish
5. Run the Edge Server.
6. Define an `ECSpec` to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec` (via `run define myECSpec ECSpec.xml`)
7. Set a JMS subscriber to the defined ECSpec.

   For example, set a JMS subscriber for `myECSpec` reports (via `run subscribe myECSpec jms:/queue/jms%2FTestQCF/jms%2FTestQ`)
8. View `JMSTest` MDB messages showing `ECReports` for the defined `ECSpec` in the `domains/domain1/logs/server.log` file of the Sun Java System Application Server installation directory.

TIBCO

TIBCO Enterprise for JMS

To run the TestQueueReceiver sample program within a TIBCO Enterprise for JMS deployment, perform the following:

1. Configure the JNDI provider and JMS server as specified in the *RFTagAware Deployment Guide*.
2. Configure RFTagAware Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `/samples/JMSSamples/TIBCO/etc` directory into the `/etc` directory of the RFTagAware installation directory.
   - Modify the `jms.options` file in the `/etc` directory of the RFTagAware installation directory with the appropriate paths for the specified environment variables
   - Modify the `naming.props` file in the `/etc` directory of the RFTagAware installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify the `edge.props` file in the `/etc` directory of the RFTagAware installation directory to set the `com.connecterra.ale.notificationDriver.jms.default.namingPropertiesFile` property to the fully-qualified file name for `naming.props`.
   - (*only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `/etc` directory of the RFTagAware installation directory to point to ALL the relevant `JMS_LIB .jar` files listed in `jms.options`. You can use either fully qualified or relative pathnames. Specify one `.jar` file per line, using the format shown below.

     For example, assume that `InstallRoot` is the root of the application server, or path to the top directory of the application server. If you are using a fully qualified pathname, you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.jar
     ```

     You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the RFTagAware installation directory. In this case you might add an entry like this to `edge.wrapper.conf`:
     ```
     wrapper.java.classpath.31=../AppServerLib/someJarFile.jar
     ```

     Create separate `wrapper.java.classpath` entries for each `JMS_LIB .jar` file listed in `jms.options`.
3. Build the `TestQueueReceiver` sample program by invoking `build.bat` or `build.sh` (set the environment variables appropriate to the build environment).
4. Run `/samples/JMSSamples/TIBCO/TestQueueReceiver.bat` (or `.sh`) to start a queue receiver for a specified queue connection factory and queue name (for example, `TestQueueReceiver TestQCF TestQ`).

5. Run the Edge Server.
6. Define an ECSpec to the Edge Server.

   For example, use the `SubscribeSample` to define `myECSpec` (via `run define myECSpec ECSpec.xml`).

7. Set a JMS subscriber to the defined `ECSpec`.

   For example, set a JMS subscriber for `myECSpec` reports (via `run subscribe myECSpec jms:/queue/TestQCF/TestQ`).

8. View `TestQueueReceiver` messages showing `ECReports` for the defined `ECSpec` in the console where `TestQueueReceiver` was started.

Chapter 8: Sample .NET Applications

Contents

This chapter describes how to use the sample .NET applications provided in your RFTagAware installation. Unlike other parts of RFTagAware, the sample applications are free for you to use and modify for your own purposes. You may use them as a starting point for developing your own applications.

Overview

There are several .NET samples provided with RFTagAware:

- `SQLNotificationSample.NET` — shows how to subscribe for delivery of ALE event cycle reports via TCP and persist the report information in an SQL database. This sample is written in VB.NET.
- `ALESample.NET` — shows how to use the ALE API to define, undefined, redefine subscribe, unsubscribe, suspend and unsuspend an event cycle specification (`ECSpec`). This sample is written in C#.
- `ALEPCSample.NET` — shows how to use the APEPC API to program tags via a cache specification (`EPCCacheSpec`) and programming cycle specification (`PCSpec`). This sample is written in C#.
- `BizTalkSample.NET` — shows how to include tag reading into a business workflow. This sample is a BizTalk Orchestration.

Setting Up Your Development Environment

To run samples, you need only the .NET runtime environment, which is available free of charge from Microsoft for the Windows operating system. You should also be familiar with the .NET SDK, SOAP interfaces, and IIS or other webservers.

For detailed information about system requirements, prerequisite software, and how to install RFTagAware, see the *RFTagAware Deployment Guide*.

Interfacing with the Edge Server

The .NET samples use Simple Object Access Protocol (SOAP) over HTTP to communicate with the RFTagAware Edge Server. SOAP is an XML based computer to computer interface that is lightweight and intended for the exchange of structured information in a decentralized, distributed environment.

WSDL Files

Web Services Description Language (WSDL) defines the "contract" between client and server in the exchange of SOAP messages. The IDE will automatically generate a proxy for you from WSDL files. This proxy will present the SOAP messages as a collection of method calls (one method per SOAP message). The result is an interface which is almost as easy to use as a 'C' DLL.

The WSDL files you need are located in the `share\schemas` subdirectory:

```
EPCglobal-ALE-1_0.wsdl
ALEPCService.wsdl
```

The associated schemas are also in `share\schemas`:

```
EPCglobal-ALE-1_0.xsd
ALEPC.xsd
```

Using a Virtual Directory

A virtual directory is a reference to a directory managed through a Web Server. You configure a Web Server to be aware of certain directories on your computer or network by defining a Web Server directory that maps to a physical one. This linkage (called a virtual directory) is how a Web Server exerts control over access to information.

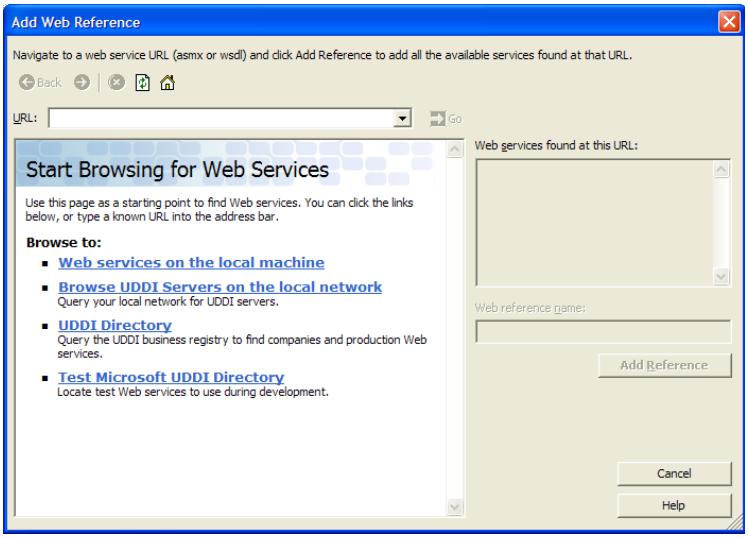
From the webserver, define a virtual directory that points to the WSDL files or place the WSDL and schema files in an existing virtual directory.

Setting Up the Microsoft IDE

You need to configure the Microsoft IDE to let applications use these interfaces to the Edge Server.

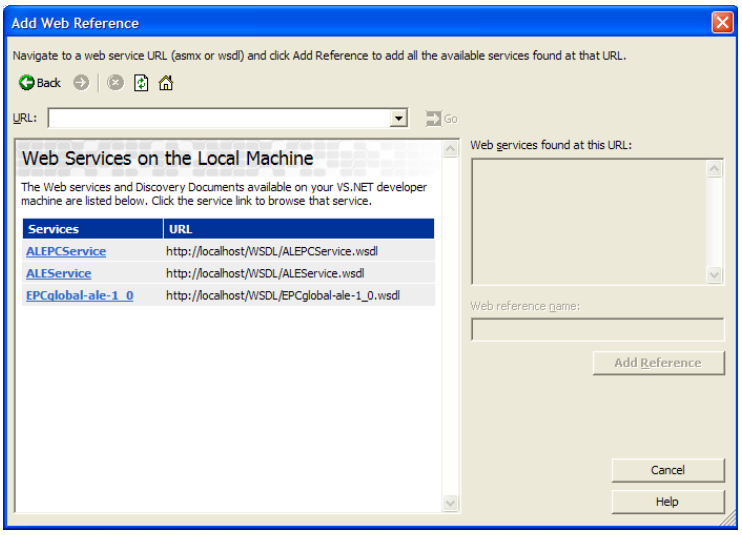
1. From the IDE, select **Project, Add Web Reference**.

   The start browsing for web services page appears.



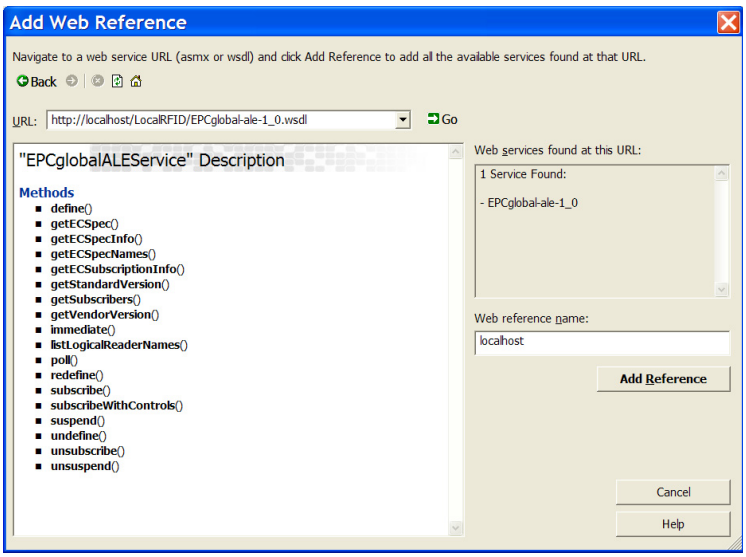
2. Click on **Web services on the local machine**.

The web services on the local machine page appears.



3. Select the service for which you want to generate a proxy. In this example we selected the `EPCglobal-ale-1_0` service. This is the service used in the `SQLNotificationSample.NET` and the `ALE.NET` samples.

   The service description page appears.



4. Click the **Add Reference** button.

   At this point, your application is ready to use the ALE SOAP interface. See the detailed sample descriptions later in this chapter for information on how to use the methods in this interface.

Using the Reader Simulator with the Samples

Before you run the samples, you may want to quickly familiarize yourself with the Reader Simulator that comes with RFTagAware. The simulator is software that emulates a **reader and writer** of tags. The Edge Server communicates with the Reader Simulator exactly as it would with a real tag reading device or real printer/tag programming device. Using the simulator, you can test your software in a controlled environment where you can model many "real world" events.

Like actual read and write devices, the simulator is passive, in that it does nothing unless instructed. There are no "start" or "stop" commands, nor are there any user configurable options. From the time you start the simulator, it is simply capable of processing requests from the Edge Server.

The simulator emulates the ThingMagic Mercury4 Reader.

This section provides a quick summary of the simulator; for more detailed information, see the *RFTagAware Deployment Guide*.

Configuring the Simulator

The default `edge.props` file that is initially installed on your system contains the following block of properties definitions for the Reader Simulator:

```
com.connecterra.ale.reader.SimReadr.class =
com.connecterra.ale.readertypes.ThingMagicMercury4PhysicalReader
com.connecterra.ale.reader.SimReadr.hostname = localhost
com.connecterra.ale.reader.SimReadr.port = 5050
com.connecterra.ale.reader.SimReadr.defaultRate = 0
com.connecterra.ale.reader.SimReadr.uhf2LogicalReaderName = ConnecTerra2
com.connecterra.ale.reader.SimReadr.uhf1LogicalReaderName = ConnecTerra1
```

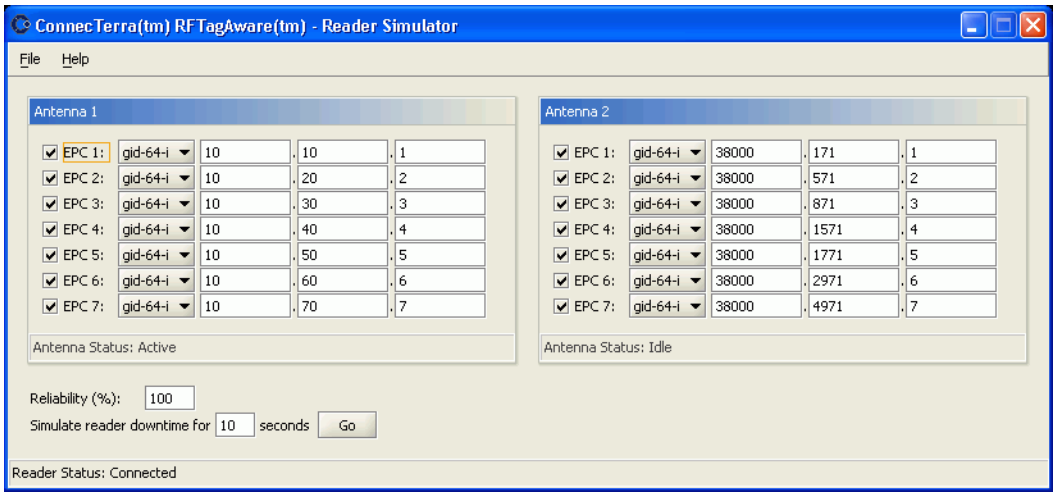
This block of properties defines a ThingMagic Mercury4 reader with two antennas running on your local system on port 5050. This set of properties actually refers to the RFTagAware Reader Simulator.

Starting the Simulator

From the `control\bin` directory, run the script:

```
RunReaderSim
```

When you run this script, the Reader Simulator appears:



By default, the simulator has two antennas. Each antenna has its own tags.

You read from and write to each antenna separately. When the Edge Server is configured as described above, the logical reader name `ConnecTerra1` refers to Antenna 1, and the logical reader name `ConnecTerra2` refers to Antenna 2.

The antenna status shows when it is being read.

Each tag has:

- A checkbox that indicates whether the tag is considered to be within range of the antenna. Unchecking this box is like moving the tag away from the antenna.
- Drop down dialog box where you can select the EPC format to emulate.
- 3-4 text boxes (depending on format) that contain the values of each of the fields of the tag's EPC code.

When reading:

- The tag values that you see in the simulator are sent to the Edge Server.

When writing:

- Uncheck **all but one tag**. There must be only one tag in the "field" of the antenna. More than one tag is an ambiguous write situation and will generate an error.
- The tag is written in the format you define in the `EPCCache` specification or replenishment command.

When either reading and writing:

- Reliability setting allows for random failures on attempted commands.
- Simulation of downtime is useful when testing read failure handling vs. no data.

Running the Samples

The samples are in the `samples` directory. Each sample has its own subdirectory:

- `samples\SQLNotificationSample.NET`
- `samples\ALESample.NET`
- `samples\ALEPCSample.NET`
- `samples\BizTalkSample.NET`

Each subdirectory contains:

- Executable(s) to run the sample.
- Source files that produced the executable(s).
- `project.sln` (and support) files for bringing up the application in the .NET IDE.
- Any scripts needed to set up the runtime environment.
- `ECSpecs` subdirectory containing the XML specification files necessary for that particular example.
- Icons and .GIF files used in building the application.

How to Run the Samples

The instructions for running all samples are the same:

1. Make sure the Edge Server is running.

   The Edge Server is installed as a service, so:

   - From the **Start** menu, select **Settings, Control Panel, Administrative Tools, Services**.
   - Double click **RFTagAware Edge Server**.
   - Click **Start**.

2. Start the Reader Simulator:

   - From the `control\bin` directory, run the script:

     ```
     RunReaderSim
     ```

3. Then, depending on which sample you want to run, follow the instructions in:

   SQLNotificationSample.NET (page 8-8)

   ALESample.NET (page 8-11)

   ALEPCSample.NET (page 8-17)

   BizTalkSample.NET (page 8-24)

SQLNotificationSample.NET

This sample shows you how to:

- Connect to the Edge Server on a specified host and port.
- Define and undefine an `ECSpec`.
- Subscribe and unsubscribe for notifications via a TCP URI.
- Specify a database server name.
- Start and stop an SQLNotification listener.

This sample demonstrates how to receive notifications from an Edge Server and persist notification data into SQL Server database. For the sake of simplicity, it does not provide extended capabilities.

Additional Requirements

- SQL Server 2000
- Create Schema privilege

How to Install

1. Navigate to the directory:
   `samples\SQLNotificationSample.NET\SQLScripts`
2. Run the script:
   `CreateDataBase.bat`

   This launches the SQL script to define a schema.

   By default, `CreateDataBase.bat` sets the host name of the SQL Server to `localhost`. If your host name is different, modify `CreateDataBase.bat` to specify the correct SQL Server.

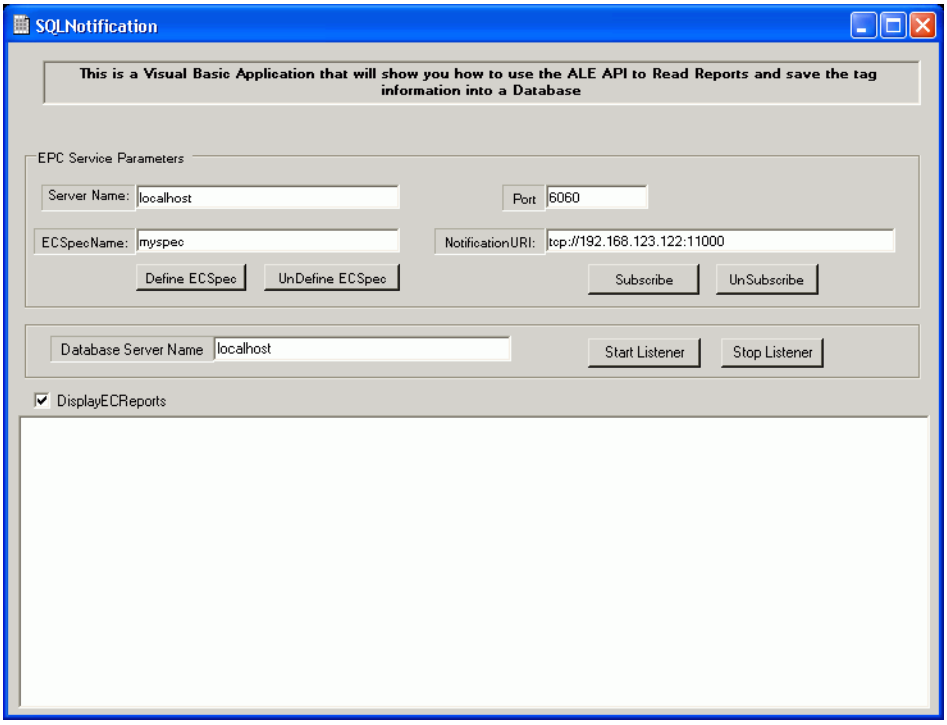
   Note: Make sure your SQL server is installed with Authentication mode as "SQL Server and Windows". (You can check this in the properties window, Security tab.) If you prefer to run this example with "Windows only" mode, you will need to modify your SQL script as well as the sample code connection string to use a trusted connection.

3. Navigate back to the directory:
   `SQLNotificationSample.NET`

How to Run SQLNotification.exe

1. From the `SQLNotificationSample.NET` directory, double click:
   `SQLNotification.exe`

   The SQL notification form appears.



2. Fill in the following fields:

   Server Name: the name of the host running the RFTagAware Edge Server from which to request reports. The default is `localhost`, meaning the same host as the one running the `SQLNotificationSample.NET` application.

   Port: The port number assigned to the Edge Server when it was installed. The default port assignment is 6060.

   ECSpecName: what you would like to call the ECSpec (any user defined name).

   NotificationURI: a string consisting of `tcp://`, followed by an IP address or name of the machine running the application. Since this URI will be used to make a TCP socket, it must refer to the local machine or the attempt will fail. The port assignment for this URI is 11000 but if this port is already being used by another application, it can be set to any unused port.

   DisplayECReports: Make sure this checkbox is selected.

3. Click **DefineECSpec**.

A dialog appears, asking you to select an event cycle (`ECSpec`) file. We provide a default, prewritten `ECSpec.xml` file, but as you get more comfortable with the sample application, you may want to experiment with changing some of the values in this file. See Chapter 5: Reading Tags Using the ALE API for information about the different values that you can specify in an `ECSpec`.

4. For now, choose the prewritten `ECSpec` XML file:
   `ECSpecs\ECSpec.xml`
5. Click the **Subscribe** button, followed by the **Start Listener** button. You should see:
   - `ECReports` being generated every 2 or 3 seconds. These reports contain tags that the Edge Server got from the Reader Simulator.
   - SQL Database being populated. To check this:

     Start SQL Server Enterprise Manager.

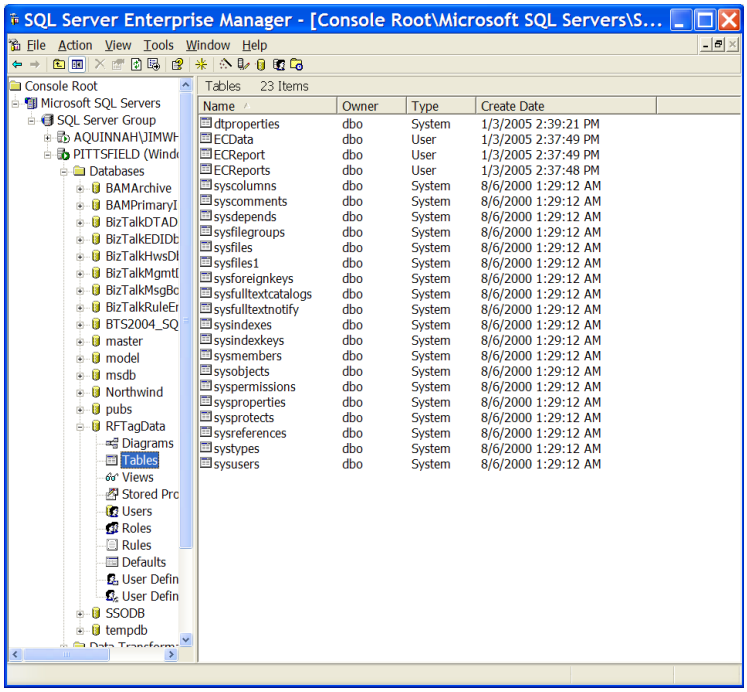
     Open Server Groups.

     Open your SQL Server.

     Open Databases.

     Open RFTagData.

     Open ECReports Table, select rows. The amount of data in a table depends on the kinds of reports received. ECReports Table logs each report captured. It will be the more populated of the tables.



Programming Notes

This application is meant as a teaching tool, and is provided "as is." To get your development environment set up to work with this sample:

- Install and configure Visual Studio.
- Double click `SQLNotification.sln`. The IDE is launched with all source files

The `SQLNotificationSample.NET` sample has following 4 components,

- Edge Server web service proxy.

  Edge Server proxy/stub class that allows this sample application to talk to the Edge Server for defining and subscribing notifications. This functionality is in the file:
  `EDgeServerWebserviceStub\Web References\localhost\Reference.cs`

- User interface window.

  Simple Windows form that captures user parameters like Edge Server name and port, database server, and TCP notification URI. It allows the user to subscribe to an `ECReports` instance and start/stop listening for notifications as well as see message details. This form actually starts up the TCP listener on separate thread when the user clicks the **Start Listener** button. This functionality is in the file:
  `SQLNotificationForm.vb`

- TCP Listener.

  This simple socket handler object starts a server socket for TCP notifications from the Edge Server and starts listening on the socket for notifications. When notifications are received, it fires up new instances of `SQLNotificationHandler` to handle persistence in separate threads. This functionality is in the file:
  `TCPListener.vb`

- Notification data extractor.

  The `SQLNotificationHandler` class is designed to handle notifications and persist the data from these notifications into database tables according to the table schema. This functionality is found in the file:
  `SQLNotificationHandler.vb`

ALESample.NET

This sample shows you how to use the ALE API to read EPC tags. Specifically, it shows you how to use the ALE API to define, undefine, redefine subscribe, unsubscribe, suspend and unsuspend an event cycle specification (`ECSpec`).

Additional Requirements

None

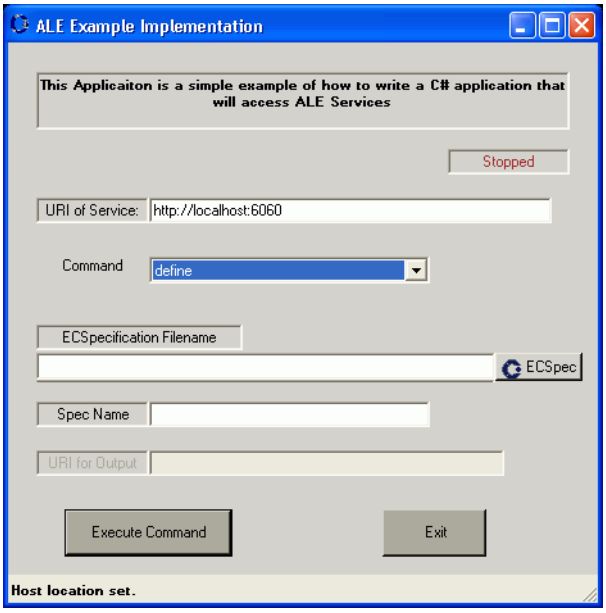
How to Install

No additional installation steps necessary.

How to Run ALESample.NET

1. Navigate to the directory:
   `samples\ALESample.NET`
2. Double click:
   `ALE.NET.exe`

   The `ALESample` GUI appears.



   The application will walk you through the instantiation and use of an `ECSpec` instance. This application allows you to define an `ECspec` and sign up for the ensuing notifications. The **Command** drop down menu selects a method of the ALE API. Other fields below the **Command** menu let you specify arguments to the specified method (which fields are available depend on which command you select). After you specify arguments, the **Execute Command** button sends the command to the Edge Server.

3. The first step is to define the `ECspec`. In the **Command** drop down menu, select **define** (default first command).
4. Click the **ECSpec** button.

This opens a dialog box where you can navigate to the XML file that contains the definition of your `ECSpec`. In this example, navigate to `ALESample.NET/ECSpecs` and select the predefined XML file: `ECSpec.xml`.

5. With a text editor, open the `ECSpec.xml` file you just selected. Here are some excerpts from this file:

ImmediateProgramSample Run Output

```
<?xml version="1.0" encoding="UTF-8"?>
<ale:ECSpec targetNamespace="urn:epcglobal:ale:xsd:1"
xmlns:ale="urn:epcglobal:ale:xsd:1"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    xmlns:aleext="http://schemas.connecterra.com/EPCglobal-extensions/ale"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    creationDate="2004-11-15T16:18:43.500Z"
    schemaVersion="1.0"
    includeSpecInReports="false" >

<logicalReaders>
   <logicalReader>ConnecTerra1</logicalReader>
</logicalReaders>

<boundarySpec>
   <repeatPeriod unit="MS">10000</repeatPeriod>
   <duration unit="MS">2000</duration>
</boundarySpec>

<reportSpecs>
   <reportSpec reportName="SubscribeSample Report" reportIfEmpty="true">
   <reportSet set="CURRENT" />
   <output includeCount="true"
                   includeEPC="false"
                   includeRawDecimal="false"
                   includeRawHex="false"
                   includeTag="true"  />
   </reportSpec>
</reportSpecs>
<aleext:applicationData>application-specific data here
</aleext:applicationData>

</ale:ECSpec>
```

We specify the logical reader as `ConnecTerra1`, which is mapped to "Antenna 1" in the Reader Simulator by default. We also specify that our event cycle is to be 2 seconds long (2000 MS), repeated each 10 seconds (10000 MS). The final section defines a report specification that requests both a count and a list of all the `CURRENT` tags visible to logical reader `ConnecTerra1`.

As you get more comfortable with this sample, you can experiment with changing the values in `ECSpec.xml`. For now, use the default values.

6. Now, returning to the ALE sample GUI, press the **Execute Command** button.

This invokes the `define` method, which now creates an `ECSpec` object based on the information we provided in `ECSpec.xml`.

Defining an `ECSpec` is NOT the same as activating it. You have not yet told a reader to read any tags or done anything else with the `ECSpec` yet. You have simply defined a list of actions (read cycles, delivery activities, and so on) that can take place some time in the future once the `ECSpec` *is* activated by a method such as `poll` or `subscribe`.

7. Take a look at the status bar at the bottom of the form. Note that the status changes to **Defined**.

8. From the command drop down menu, select **Subscribe**.

   We now need to define a location where we can receive reports. For now, write to a file by typing the following in the **URI for Output** box:
   `file:///C:/Temp/ECSpec.out`

   Note forward slashes and no spaces in pathname.

9. Press the **Execute Command** button.

   The `ECSpec` we defined is now being activated by the Edge Server.

10. Take a look at the `ECSpec.out` file we specified in the **URI for Output** box.

    - Recall that the `ECSpec` we defined said to report on tags every 10 seconds — note that the file is being updated every 10 seconds.
    - Compare the tag entries in `ECSpec.out` with the values shown in the Reader Simulator. You can see that the Edge Server is obtaining the tag values from the simulator, and reporting these values in the reports it writes to `ECSpec.out`.

      Feel free to change the values on the simulator and inspect the corresponding report.

11. Now let's use the ALE Sample GUI to look at `ECspec` information. From the **Command** drop down menu, select **get spec info**.

12. Press the **Execute Command** button.

    Note that the form now displays the (active) status of the `ECspec` we defined.

13. Try the suspend command — from the **Command** drop down menu, select **suspend**, then press the **Execute Command** button.

    This tells the Edge Server to suspend the `ECSpec`.

14. Retry the get spec info command — from the **Command** drop down menu, select **get spec info**, then press the **Execute Command** button.

    Notice that the `ECSpec` is now suspended (not reporting). Unsuspending will re-enable the notifications

15. To close the application:

    - Select and execute **Unsubscribe**.
    - Select and execute **Undefine**.
    - Application status should say **Stopped**.

The Edge Server will continue to produce reports until it is told to stop via the unsubscribe command. This is true even across restarts of the Edge Server. Therefore, be sure to unsubscribe your `ECSpec` when you end the sample application. If you forget to do that, the application will remind you on exiting.

Programming Notes

The `ECExample` class (`ECExample.cs`) handles all events from the main GUI. Most events have to do with gathering data from the form and saving data into private member variables or assisting the user in entering data into the form. There are two types of assistance:

- Helping the user find a file (done via the standard File Open Dialog that all Windows applications use).
- Helping the user generate a name for the `ECSpec` (parsing out the body of the XML filename as the `ECSpec` name).

With all of these event processing methods, the flow is disconnected. The process flow begins with them being called, they perform a small amount of work and then they return.

The flow of execution really starts when the user presses the **Execute Command** button. When the user presses this button Windows calls the `OKButton_Click` method. This is largely a long case statement that switches on `Command`.

The processing of each command involves communication with the service layer class of the applet called `ALEServiceFacade (ALEServiceFacade.cs)`. The appropriate properties on the form are given to the `ALEServiceFacade` class via its `set` methods and the command is executed via the appropriate `do` method. The `ALEServiceFacade` class contains `set` methods for `ECSpec` (name and file), `NotificationURI`, and `ServiceURI` (Location of the Edge Server). There is a `do` method for each supported command.

The `ALEServiceFacade` class communicates back to the `ECExample` class in one of three ways:

- If a callback is defined when the class is constructed, status information will be send back to this callback. This is the way `ALEServiceFacade` is "new'ed" so status information is returned which is displayed on the status line appearing at the bottom of the main GUI.
- If there is an error, an exception is thrown. Exceptions from the SOAP layer are deliberately not caught by `ALEServiceFacade` and allowed to go back to the GUI class for processing (the only exception is when a file stream needs to be closed in which case it is caught, the stream closed and the exception re-thrown). This error is displayed in the status bar prefaced by **Failed** and then the message in the exception (NOTE: the SOAP layer does not supply any message text, the description of the failure is found in `SoapException_instance.Detail.FirstChild.InnerText`.
- There is a return value from the method that is largely useless (almost always true) unless the method is a request for a status. In this case a string is returned that contains the status information.

If a status string is returned from the `ALEServiceFacade` class, a second form is evoked that presents that data to the user. This is a very simple form that has a text box containing the status information and a **Done** button.

Using the ALE API

The `ALEServiceFacade` class is the class that actually used the SOAP proxy as generated by the IDE. This class is only instantiated once by this application but the concept behind this class is that an instance can be instantiated for every active `ECSpec`. When this class is constructed, the constructor creates an instance of the `EPCglobalALEService` object. `EPCglobalALEService` is the SOAP proxy. Its methods are equivalent to the ALE API methods described in ALE: Main Tag Reading Interface on page 5-3.

The first step the user performs in the application is to provide an `ECSpec` to define the logical group of readers and their properties. We read this in the `setSpec` method via the `XmlSerializer` class. Note the explicit namespace of `urn:epcglobal:ale:xsd:1`. This is required in order to correctly resolve all the references in the XML. This builds our `ECSpec` object. This object has all the information from our original XML file — logical reader name, boundary conditions, report specifications, etc.

Next we have a series of "do" methods. Each method is named `doAction` where `Action` is the ALE command that the user selected in the GUI command drop down menu. Each time you press the **Execute Command** button in the GUI, the "do" method corresponding to the selected command is called.

The act of sending the SOAP message to RFTagAware is very similar from method to method. We will walk through the `doDefine` method to show how this is done. The `doDefine` method sends our newly created `ECSpec` object to the Edge Server. The arguments for the ALE `define` method are encapsulated in an object called `Define` which the proxy generator builds. This object contains the arguments that will be passed to the Edge Server. The `define` method requires an `ECSpec` object and a name for that object. If you inspect the `Define` object, you will see that it is largely just a struct with two strings — `spec` and `specname`. Set both those names with an appropriate value and call the `service.define` method and you have just made a call to the Edge Server.

Note the exception handling that is done in this method. If there is a `SoapException`, the application will try to determine if the problem is that a pre-existing definition of the same name exists and will synchronize the application with what the Edge Server is doing via the `syncWithService()` method. Note that error text from the Edge Server is not contained in the `exception.message` member but rather in `SoapException.Detail.FirstChild.InnerText`.

Now that the Edge Server is aware of our newly created `ECSpec` object, we can use other "do" methods to read tags following the `ECSpec`'s instructions. For example, `doSubscribe` subscribes notifications to a URI. `doSubscribe` behaves similarly to `doDefine`. It first creates a `Subscribe` object, then populates that object with relevant parameters — the name of an `ECSpec` and a notification URI. Finally, it invokes the relevant ALE API method from the `EPCglobalALEService` object (in this case, `subscribe`) to communicate with the Edge Server.

The other "do" methods behave similarly — they create their own object, populate the object with relevant parameters, then invoke an ALE API method from the `EPCglobalALEService` object to communicate with the Edge Server.

ALEPCSample.NET

This sample shows you how to use ALEPC API to write EPCs onto tags. It uses the following ALEPC API calls:

- Define Cache
- Define PC Specification
- Subscribe Cache
- Subscribe PC Specification
- Poll
- Cache Information
- List Caches
- List PC Specifications
- PC Specification Information
- Deplete Cache
- Replenish Cache
- Unsubscribe Cache
- Unsubscribe PC Specification
- Undefine PC Specification
- Undefine Cache

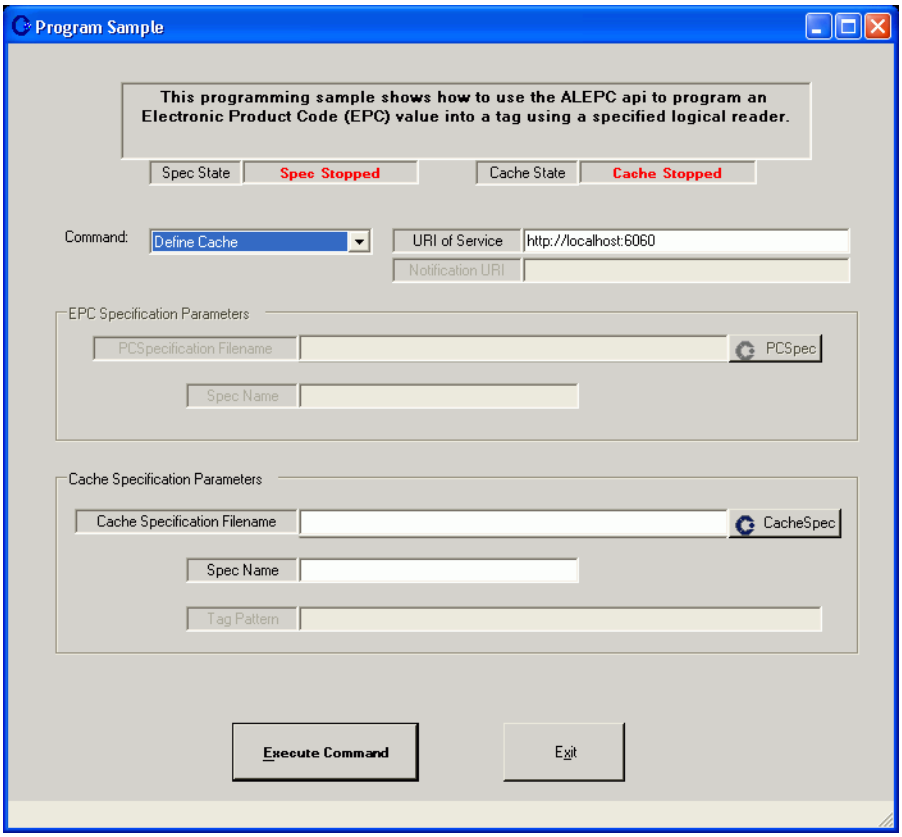
Additional Requirements

None.

How to Run ALEPC.NET

1. Navigate to the directory:
   `samples\ALEPCSample.NET`
2. Double click:
   `ALEPC.NET.exe`

The `ALEPCSample` GUI appears.



This application programs tags via the Edge Server. It is intended as a teaching aid. To that end the interface has been oriented to a single thread of execution. The user will define an `EPCCacheSpec` and a `PCSpec`, sign up for notifications via `subscribe` (`EPCCacheSpec`) and `subscribe` (`PCSpec`), and then (optionally) the user can look at a series of reports to see how they are viewed by the Edge Sever (list specs, list caches, get cache information, get `PCSpec` information).

The tag programming is done via the `poll` command where the results of the programming attempt are shown in a results window.

Note that with each command selected, only the fields legal for that command will accept input.

To provide a mechanized writing capability, this application defines an `EPCCacheSpec` and a `PCSpec` that give instructions on how to write tags in a repetitive fashion.

3. The first thing to do is define a cache via a cache spec. From the **Command** drop down menu, select **Define Cache** (default first selection).

4. Click the **CacheSpec** button.

   This opens a dialog box where you can navigate to the XML file that contains the definition of your cache spec. In this example, navigate to the subdirectory:
   `ALEPCSample.NET/ECSpecs`

and select the predefined XML file:
`CacheSpec.xml`

5. With a text editor, open the `CacheSpec.xml` file you just selected. Here are some excerpts from this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<EPCCacheSpec xmlns="http://schemas.connecterra.com/alepc">
<applicationData>application-specific data here</applicationData>
<threshold>10</threshold>
<includeCacheContent>true</includeCacheContent>
</EPCCacheSpec>
```

   This cache spec specifies a low-cache reporting threshold of 10. This means that whenever a cache is below its reporting threshold, it issues low-cache reports to its subscribers. Also note that the `EPCCacheReport` instances associated with this cache will include a description of the current cache contents (true), not just the count of the remaining cache entries (false).

6. Now, returning to the ALEPC sample GUI, press the **Execute Command** button.

   Note that the status line at the bottom of the application now indicates that the cache is defined.

7. Next, define a PC Specification — from the **Command** drop down menu, select **PCSpec Command**.

8. Click the **PCSpe**c button.

   This opens a dialog box where you can navigate to the XML file that contains the definition of your `PCSpec`. In this example, navigate to the subdirectory:

   `ALEPCSample.NET/ECSpecs`

   and select the predefined XML file:

   `PCSpec.xml`

9. With a text editor, open the `PCSpec.xml` file you just selected. Here are some excerpts from this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<PCSpec xmlns="http://schemas.connecterra.com/alepc">
<cacheName>CacheSpec</cacheName>
<applicationData>application-specific data here</applicationData>

   <logicalReaders>
      <logicalReader>ConnecTerra2</logicalReader>
   </logicalReaders>

   <boundarySpec>
      <trials>1</trials>
      <duration>4000</duration>
   </boundarySpec>
</PCSpec>
```

   This `PCSpec` indicates that we will call the cache we just defined by the name `CacheSpec`. Note that this is the name of the *programming object* that represents the cache of EPC values — it has

no relation to the XML file name that *initially* contained information about the cache. From here on out, we refer to the cache by its programming object name, `CacheSpec`.

We will write the tag using logical reader `ConnecTerra2`, which maps to the Reader Simulator's Antenna 2. We will attempt to write the tag only once. We will spend, at most, 4 seconds (4000 MS) retrying failed tag writing operations.

10. Returning to the ALEPC sample GUI, press the **Execute Command** button.

    Note that the status line at the bottom of the application now indicates that the `PCSpec` is defined.

11. Next, subscribe to notification events:

    - In the Command drop down menu, select **Subscribe Cache**.
    - In the **Notification URI** field, type
      `console:///`
    - Click the **Execute Command** button.
    - In the Command drop down menu, select **Subscribe PCSpec**, then press the **Execute Command** button.

12. Now we can start programming tags — in the Command drop down menu, select **Poll**, then click the **Execute Command** button.

    This command produces an error. Examining the report shows that our cache has no tags:
    `<status>CACHE_EMPTY</status>`

13. We need to replenish the stock of EPC values for tags — in the Command drop down menu, select **Replenish**.

14. In the **Tag Pattern** box, define 300 tags by typing in the pattern:
    `urn:epc:pat:sgtin-64:3.0036000.5.[1-300]`

15. Click the **Execute Command** button.

16. Check on the replenishment — in the Command drop down menu, select **Cache Information**, then click the **Execute Command** button.

    Note that the report indicates that our cache now has 300 tags:
    `<cacheSize>300</cacheSize>`

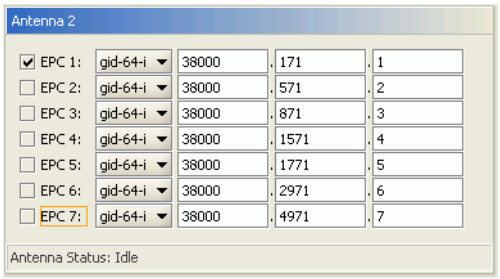
17. Click the **OK** button.

18. Now we can try our tag writing operation again — in the Command drop down menu, select **Poll**, then click the **Execute Command** button.

    This command also produces an error.
    `<status>MULTIPLE_IN_FIELD</status>`

    The logical reader `ConnecTerra2` sees too many tags in its field, and it does not know which one to write to. In order to write a tag, it must see only ONE tag in its field.

To correct this error, go to the Reader Simulator and deselect **all but one** tag checkbox for Antenna 2 (logical reader `ConnecTerra2` maps to Antenna 2).



19. Try once again — in the Command drop down menu, select **Poll**, then click the **Execute Command** button.
    `<status>SUCCESSFUL</status>`

    Take a look at the tag value we wrote:
    `<epc>urn:epc:tag:sgtin-64:3.0036000.000005.5</epc>`

20. Write another tag and look at its EPC value:
    `<epc>urn:epc:tag:sgtin-64:3.0036000.000005.6</epc>`

21. If we kept writing all the tags we defined in the replenishment, we would eventually reach the low threshold value specified in the `CacheSpec` (10 tags left). At this point we would receive a report notifying us of the low threshold status.

22. Now deplete tags so they can be used later — in the Command drop down menu, select **Deplete Cache**, then click the **Execute Command** button.

    Running the **Cache Information** command will verify they are gone.

23. Now "unwind" out of the application:

    - Execute **Unsubscribe PCSpec**.
    - Then **Unsubscribe CacheSpec**.
    - Then **Undefine PCSpec**.
    - Then **Undefine CacheSpec**.

    Status fields should both say stopped.

    You have successfully programmed EPC tags.

Programming Notes

This example comes with all the source and necessary project files to build in the .NET IDE. The executable is also precompiled and placed in the directory:

`\sample\ALEPCSample.NET\`

This is an overview of the samples source.

Double click on the solution file:

```
ALEPC.NET.sln
```

This solution file is loaded with all the source files.

The `PCSpecExample` class (`PCSpecExample.cs`) handles all events from the main GUI. Most of these events have to do with gathering the data from the form and saving data into private member variables or assisting the user in entering data into the form. There are two types of assistance:

- Helping the user find a file (done via the standard File Open Dialog that all Windows applications use).
- Helping the user generate a name for the file (parsing out the body of the filename as the CacheSpec or PCSpec name).

With all of these event processing methods, the flow is disconnected. The process flow begins with them being called, they perform a small amount of work and then they return.

The flow of execution really starts when the user presses the **Execute Command** button. When the user presses this button Windows calls the `ExecuteCommandButton_Click` method. This is largely a long case statement that switches on `Command`. The processing of each command involves communication with the service layer class of the applet called `ALEPCServiceFacade` (`ALEPCServiceFacade.cs`). The appropriate properties on the form are given to the `ALEPCServiceFacade` class via its `set` methods and the command is executed via the appropriate `do` method. The `ALEPCServiceFacade` class contains a set method for `CacheSpec`, `PCSpec` (name and file), NotificationURI, and ServiceURI (Location of the Edge Server). There is a `do` method for each supported command.

The `ALEPCServiceFacade` class communicates back to the `PCSpecExample` class in one of three ways:

- If a callback is defined when the class is constructed, status information will be send back to this callback. This is the way `RFIDPC` is "new'ed" so status information is returned which is displayed on the status line appearing at the bottom of the main GUI.
- If there is an error, an exception is thrown. Exceptions from the SOAP layer are deliberately not caught by `ALEPCServiceFacade` and allowed to go back to the GUI class for processing.
- There is a return value from the method that is largely useless (always true) unless the method is a request for a status. In this case a string is returned that contains the status information.

  If a status string is returned from the `ALEPCServiceFacade` class, a second form is evoked that presents that data to the user. This is a very simple form that has a text box containing the status information and an **OK** button.

  Note: The applet does not maintain its state information across restarts. It will assume you are in a stopped state when you start the applet. If you left a `CacheSpec` or `PCSpec` active in the Edge Server when you exited the applet, the two applications will be out of sync (the attempt to define your `CacheSpec` or `PCSpec` will be rejected by the Edge Server and the application will attempt to determine the appropriate state).

The **Stop** command can be used to undefine the specification, regardless of the state.

Using the ALEPC API

The `ALEPCServiceFacade` class is the class that actually used the SOAP proxy as generated by the IDE. It is very similar to the `ALEServiceFacade` class in the `ALE.NET` example (see ALESample.NET on page 8-11). This class is only instantiated once by this application but the concept behind this class is that an instance can be instantiated for every active `ECSpec`. When this class is constructed, the constructor creates an instance of the `ALEPCService` object. `ALEPCService` is the SOAP proxy. The methods are equivalent to the ALEPC API methods described in ALEPC: Main Tag Writing Interface on page 6-3.

The first step the user performs in the application is to provide a `CacheSpec` to define the properties (behavior) of a collection of tag values and how they will be assigned. We read this in the `setCacheSpec` method via the `XmlSerializer` class. (Note the additional namespace of `http://schemas.connecterra.com/alepc`) This builds our `CacheSpec` object.

Next we have a series of "do" methods. Each method is named `doAction` where `Action` is the ALEPC command that the user selected in the GUI command drop down menu. Each time you press the **Execute Command** button in the GUI, the "do" method corresponding to the selected command is called.

The act of sending the SOAP message to RFTagAware is very similar from method to method. We will walk through the `doDefine` method to show how this is done. The `doDefine` method sends our newly created `PCSpec` object to the Edge Server. The arguments for the ALEPC `define` method are encapsulated in an object called `Define` which the proxy generator builds. This object contains the arguments that will be passed to the Edge Server. The `define` method requires a `PCSpec` object and a name for that object. If you inspect the `Define` object, you will see that it is largely just a struct with two strings — `spec` and `specname`. Set both those names with an appropriate value and call the `service.define` method and you have just made a call to the Edge Server.

Note the exception handling that is done in this method. If there is a `SoapException`, the application will try to determine if the problem is that a pre-existing definition of the same name exists and will synchronize the application with what the Edge Server is doing. Note that error text from the Edge Server is not contained in the `exception.message` member but rather in `SoapException.Detail.FirstChild.InnerText`.

Now that the Edge Server is aware of our newly created `PCSpec` object, we can use other "do" methods to write tags following the `PCSpec`'s instructions. For example, `doSubscribe` subscribes notifications to a URI. This enables this `PCspec` to subscribe to a cache and receive notification information from the Edge Server at the address defined in the `notificationURI` parameter

`doSubscribe` behaves similarly to `doDefine`. It first creates a `Subscribe` object, then populates that object with relevant parameters — the name of a `PCSpec` and a notification URI. Finally, it invokes the relevant ALEPC API method from the `ALEPCService` object (in this case, `subscribe`) to communicate with the Edge Server.

The other "do" methods behave similarly — they create their own object, populate the object with relevant parameters, then invoke an ALEPC API method from the `ALEPCService` object to communicate with the Edge Server.

BizTalkSample.NET

This sample shows you how to use BizTalk to:

- Connect to an Edge Server inside your orchestration.
- Define an event cycle specification (`ECSpec`).
- Subscribe for notification via file URI.
- Receive `ECReport` notifications in two forms:
  - XML file
  - SQL database persistence
- Unsubscribe notification.
- Undefine the `ECSpec` for clean up.

Additional Requirements

- BizTalk
- Visual Studio
- Database schema creation/access privilege

How to Install

1. Make sure the Reader Simulator and Edge Server are both started. (From a command line in the `control\bin` directory, run the scripts `RunReaderSim.bat` and `RunEdgeServer.bat`).
2. Navigate to the directory:
   `samples\BizTalkSample.NET\ConsumeConnecterraEdgeService`
3. Run the script:
   `Setup.bat`

   This creates a database table for the `ECReport` notification, build solution, deploy, bind and start orchestrations.

This script will bind SQLNotificationPort to use a trusted connection to the database. Therefore, make sure you are currently logged in with database access privileges. If you would like to use the script-generated SQL account `rogerdb/rogerdb`, change the connection string of the SQLNotificationPort send port manually. During the solution building step, if you see XSD-related warnings or errors, disregard them as long as the solution builds successfully.

This script will also create the directories `In`, `Out`, `ECReportDrop`, and `ECReports` under `ConsumeConnecterraEdgeService`. These directory names are defaults, and may be changed by editing the `Setup.bat` file and re-running it.

4. By default, the script configures the binding file `ConsumeEdgeServiceBinding.xml` with the Edge Server URL: `http://localhost:6060/axis/services/EPCglobalALEService`

   If your Edge Server is running on a different port, change the port in this URL to point to the correct Edge Server. For this example, the Edge Server must be running on a local machine.

5. Copy the file `DefineECSpecArguments.xml` to the `In` directory under `ConsumeConnecterraEdgeService`. (Do not move it because it will be picked up by the orchestration.) This file contains sample `ECSpec` and `ECReport` notification information that is used for orchestration.

6. With a text editor, open the `DefineECSpecArguments.xml` file you just copied. Here are some excerpts from this file:

DefineECSpecArguments for BizTalkSample.NET

```
<?xml version="1.0" encoding="UTF-8"?>
<n:DefineECSpecArguments xmlns:n="http://
ConsumeEdgeServiceSchemas.DefineECSpecArgs">
<ECSpecName>myspec</ECSpecName>
<ECSpec creationDate="2004-11-15T16:18:43.500Z" schemaVersion="1.0"
includeSpecInReports="false">
<logicalReaders>
   <logicalReader>ConnecTerra1</logicalReader>
</logicalReaders>

<boundarySpec>
   <duration unit="MS">2000</duration>
</boundarySpec>

<reportSpecs>
   <reportSpec reportName="SubscribeSample Report">
   <reportSet set="CURRENT"/>
   <output includeCount="true" includeEPC="false" includeRawDecimal="false"
includeRawHex="false" includeTag="true"/>
   </reportSpec>
</reportSpecs>
</ECSpec>
<NotificationURL>[NOTIFICATIONDROPURI]</NotificationURL>
<EmptyParms/>
</n:DefineECSpecArguments>
```

We specify the logical reader as `ConnecTerra1`, which is mapped to "Antenna 1" in the Reader Simulator by default. We also specify that our event cycle is to be 2 seconds long (2000 MS). The final section defines a report specification that requests both a count and a list of all the `CURRENT` tags visible to logical reader `ConnecTerra1`.

7. Navigate to the directory:
   `samples\BiztalkSample.NET\ConsumeConnecterraEdgeService\Out`

   You should see XML message files being written to this directory. These messages tell you stage the orchestration is currently in, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<n:Result xmlns:n="http://ConsumeEdgeServiceSchemas.Result" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
ConsumeEdgeServiceSchemas.Result Result.xsd">
   <Success>Notification url is unsubscribed</Success>
</n:Result>
```

8. Navigate to the directory:
   `samples\BiztalkSample.NET\ConsumeConnecterraEdgeService\ECReports`

   You should see XML ECReport notification messages coming in.

   Each message contains an `ECReport` instance, reflecting the report specifications we set in the `DefineECSpecArguments.xml` file. Note that by default, messages arrive for a total duration of 20 seconds. This duration is set in the orchestration file, `Orchestration.odx`, located in the `ConsumeConnecterraEdgeService` directory. You can change the duration to a different value if you like.

   The figure below shows some excerpts from one of these `ECReport` message files. Note that, as specified in the `DefineECSpecArguments.xml` file, this `ECReport` includes both a count and a list of all the `CURRENT` tags visible to logical reader `ConnecTerra1`.

   Check the `ECReports` database table for `ECReport` instances. Note that the table entries contain the same information as the message files.

ECReport for BizTalkSample.NET

```
<ale:ECReports ALEID="EdgeServerID" creationDate="2005-02-11T19:42:54.500Z"
date="2005-02-11T19:42:54.500Z" schemaURL="http://schemas.connecterra.com/
EPCglobal/ale-1_0.xsd" schemaVersion="1" specName="myspec"
terminationCondition="DURATION" totalMilliseconds="2000"
xmlns:ale="urn:epcglobal:ale:xsd:1" xmlns:aleext="http://schemas.connecterra.com/
EPCglobal-extensions/ale">

<reports>
   <report reportName="SubscribeSample Report">
   <group>
   <groupList>
   <member>
   <tag>urn:epc:tag:gid-64-i:10.50.5</tag>
   </member>
   <member>
   <tag>urn:epc:tag:gid-64-i:10.10.1</tag>
   </member>
   <member>
   <tag>urn:epc:tag:gid-64-i:10.70.7</tag>
   </member>
   </groupList>
   <groupCount>
   <count>3</count>
   </groupCount>
   </group>
   </report>
</reports>

<aleext:applicationData>application-specific data here</aleext:applicationData>
<aleext:failedLogicalReaders/>
<aleext:physicalReaders>
   <aleext:physicalReader>SimReadr</aleext:physicalReader>
</aleext:physicalReaders>
<aleext:totalReadCycles>7</aleext:totalReadCycles>

</ale:ECReports>
```

9. After 20 seconds you will see a total of five messages in the directory:
   `samples\BiztalkSample.NET\ConsumeConnecterraEdgeService\Out`

   Each message contains orchestration step result details.

10. Place another `DefineECSpecArguments.xml` file in the directory:
    `samples\BiztalkSample.NET\ConsumeConnecterraEdgeService\In`

    You will see an orchestration running again for this new file.

11. Navigate to the directory:
    `samples\BizTalkSample.NET\ConsumeConnecterraEdgeService`

12. Run the script:
    `Cleanup.bat`

    This uninstalls the sample from the BizTalk server.

Index

N

P

R

S

T

W

X