BEA WebLogic RFID Edge Server™

Programming with the ALE and ALEPC APIs

Version 2.1
Revised: June 29, 2006

Contents

1. Introduction and Roadmap

Document Scope and Audience 1-1

Guide to This Document 1-2

Related Documentation 1-2

EPCglobal Standards Compliance 1-3

Using RFID Samples to Develop Applications 1-3

2. Reading and Writing Tags

Overview of the ALE API and ALE Operation 2-1

BEA Implementation of the ALE API 2-2

Benefits of the BEA Implementation 2-3

Programming Methods 2-4

Reading Tag Data 2-4

- Read Cycles and Event Cycles 2-5
- Smoothing Read Cycles with Transient Filtering 2-7
- How Applications Interact with the Edge Server ALE Engine 2-7
- ECSpec Reports 2-8

Writing Tag Data 2-9

- Programming Cycles 2-9
- Reader Implementation of Programming Cycles 2-11
- EPC Caches 2-12
- Creating Tag Caches 2-12

PCSpec Reports 2-14
Comparison of Event Cycles and Programming Cycles 2-14

3. Asynchronous Notification Mechanisms

Overview of Asynchronous Notification Mechanisms 3-1
Encoded XML Through HTTP POST 3-2
Encoded XML Through TCP Socket 3-2
Encoded XML in JMS Message 3-3
Examples of Report Delivery by Using XML with JMS 3-5
XML Written to a File 3-6
XML Displayed on the Edge Server Console 3-7
XML Sent to a Workflow Module 3-8

4. Reading Tags by Using the ALE API

Overview of the ALE API Implementation 4-2
ALE: Main Tag Reading Interface with UML Diagrams 4-3
State Diagram 4-6
Primary ALE API Data Types 4-7
ECSpec Data Type 4-7
ECBoundarySpec 4-10
ECBoundarySpec Implementation Notes 4-12
ECTime 4-12
ECTimeUnit 4-13
ECTrigger 4-13
ECReportSpec 4-13
ECReportSpec Implementation Notes 4-15
ECReportSetSpec 4-16
ECFilterSpec 4-16

EPC Patterns 4-17

ECGroupSpec 4-18

About Group Reports 4-18

Examples of Pattern URIs Used as Grouping Patterns 4-19

ECReportOutputSpec 4-22

ECReports Data Type 4-23

ECTerminationCondition 4-25

ECReport 4-26

ECReportGroup 4-27

ECReportGroupList 4-27

ECReportGroupListMember 4-28

ECReportGroupCount 4-30

ECReportGroupListMemberMemory 4-30

Other ALE API Types: BEA Extensions. 4-31

ECSpecInfo (WebLogic RFID Edge Server Extension) 4-31

ECSubscriptionInfo (WebLogic RFID Edge Server Extension) 4-32

ECSubscriptionControls (WebLogic RFID Edge Server Extension) 4-33

XML Representations 4-34

ECSpec - Example 4-35

ECReports - Example 4-35

Using the ALE Tag Reading API from Java 4-37

Using XML Serializers and Deserializers from Java 4-37

Gen2 Read Support 4-38

includedMemory. 4-38

getMemoryItem 4-39

5. Writing Tags by Using the ALEPC API

Overview of the ALEPC API Implementation 5-2

ALEPC: Main Tag Writing Interface with UML Diagrams 5-3
PCSpec 5-7
    PCSpecInfo 5-9
    PCSubscriptionControls 5-10
    PCSubscriptionInfo 5-10
    AccessSpec 5-11
        OpSpec 5-12
        DataSpec 5-16
PCWriteReport 5-17
    AccessReport 5-19
    TagReport 5-20
    OpReport 5-20
    PCStatus 5-21
    PCTerminationCondition 5-23
EPCCacheSpec 5-23
    EPCCacheReport 5-24
    EPCCacheSpecInfo 5-25
    EPCPatterns 5-26
XML Representations 5-27
    PCSpec - Example 5-28
    PCWriteReport - Example 5-29
    EPCCacheSpec - Example 5-29
    EPCCacheReport - Example 5-30
    XML Schema for PCSpec, PCWriteReport, EPCCacheSpec, and EPCCacheReport5-30
Using the ALEPC Tag Writing API from Java 5-30
    Using XML Serializers and Deserializers from Java 5-31
BEA Gen2 Write Support 5-31
    Extended API Support 5-32

Multiple Tags in Field 5-36
Parameter Substitutions 5-36
Gen2 PCSpec Examples 5-37
readEPCBank.xml 5-37
readAbsolute.xml 5-38
readPassword.xml. 5-38
writePassword.xml 5-39
writePasswords.xml 5-39
writeTagMemory.xml 5-40
writeTagMemory.xml and PCWriteReport.xml 5-40
kill.xml 5-41
lock.xml 5-42
stackLight.xml 5-42

6. Sample Java Applications

Overview of Sample Java Applications. 6-2
Setting Up Your Development Environment 6-2
Compiling and Running the Samples 6-2
ImmediateSample: Getting Started Reading Tags. 6-3
Using ImmediateSample with the Administration Console 6-6
ImmediateSample: Event Cycles and Reliability 6-7
ImmediateSample: Reading from Different Readers 6-8
SubscribeSample: Exploring Asynchronous Event Cycle Delivery 6-8
SubscribeSample Command Line Options. 6-11
ImmediateProgramSample: Writing Tags 6-12
Using ImmediateProgramSample with the Reader Simulator 6-14
ProgrammingSample: Exploring Programming Cycles and EPC Caches. 6-16
JMS Samples 6-21

Workflow Sample XML Files. 6-23

Index

CHAPTER 1

Introduction and Roadmap

The following sections describe the scope and organization of this document, *Programming with the ALE and ALEPC APIs*:

- "Document Scope and Audience" on page 1-1
- "Guide to This Document" on page 1-2
- "Related Documentation" on page 1-2
- "EPCglobal Standards Compliance" on page 1-3
- "Using RFID Samples to Develop Applications" on page 1-3

Document Scope and Audience

This programming guide describes how to use the BEA implementation of the ALE API specification to develop applications that create EPC tag-reading requests in the form of Event Cycle Specifications (ECSpecs). The guide also describes how to use the BEA ALEPC API to develop applications that create EPC tag-writing requests in the form of Programming Cycle Specifications (PCSpecs).

The ALE API is defined by The Application Level Events (ALE) Specification Version 1.0 and by the extensions to that specification provided by BEA. ECSpecs are defined by the EPCglobal ALE specification. The ALEPC API is a BEA-defined interface for writing tag data.

The guide documents the API defined by the specification and the value-added extensions provided by BEA. A prerequisite for using this programming guide is a thorough understanding of the ALE specification, which in addition to defining the API provides necessary background information.

Although the intended audience for this guide is primarily application programmers, administrators might find the general descriptions of the ALE interface useful.

Guide to This Document

- This chapter, Introduction and Roadmap, describes the organization of this document.
- Reading and Writing Tags provides a general overview of how tags are read and written through the ALE API. The chapter introduces the concepts and terminology used throughout this guide.
- Asynchronous Notification Mechanisms describes supported mechanisms for delivering reports.
- Reading Tags by Using the ALE API describes the methods available for reading tag data.
- Writing Tags by Using the ALEPC API describes the methods available for writing tag data.
- Sample Java Applications describes how to set up, run, and work with the bundled examples, which are an optional part of the installation process. You can use these applications as a starting point for developing your own applications.

Related Documentation

This document is a part of the WebLogic RFID Edge Server documentation set. The other documents are:

- *Installing WebLogic RFID Edge Server* describes how to install and configure WebLogic RFID Edge Server.
- *Using the RFID Edge Server Administration Console* is online help that describes how to use the RFID Administration Console GUI to configure ECSpecs, ECReports, RFID devices, filters, and workflows.
- *Using the Reader Simulator* describes how to use the reader simulator software included with RFID Server. The Reader Simulator minimally simulates a ThingMagic Mercury4 RFID reader.

- *RFID Reader Reference* describes how to configure the RFID devices supported by the RFID Server.
- *RFID Workflow Reference* describes how to configure and use the workflow modules included with the WebLogic RFID Edge Server.
- *ALE and ALEPC Javadoc* provides reference documentation for the ALE and ALEPC packages that are provided with WebLogic RFID Edge Server.
- The Application Level Events (ALE) Specification Version 1.0
- EPCglobal EPC Tag Data Standard Version 1.1 rev 1.27

EPCglobal Standards Compliance

WebLogic RFID Edge Server is compliant with all relevant EPCglobal standards:

- WebLogic RFID Edge Server works with RFID readers that implement the following protocols:
  - EPCglobal Class 0, Class 0+, and Class 1 RF Protocols
  - EPCglobal UHF Class 1 Gen 2 Tag Protocol (ISO 18000-6C)
  - ISO 15693 RF Interface protocol
  - ISO 18000-6B RF Interface protocol
- WebLogic RFID Edge Server supports EPC Tag Data Standard Version 1.1 rev 1.27. 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.
- WebLogic RFID Edge Server includes components that collect and filter tag data as defined within the EPCglobal Architecture Framework. WebLogic RFID Edge Server fully implements the The Application Level Events (ALE) Specification Version 1.0, which is the standard interface to filtering and collection as defined by EPCglobal (replacing earlier “Savant” specifications). WebLogic RFID Edge Server provides extensions in the areas of reader management, application integration, and tag writing.

Using RFID Samples to Develop Applications

If you install the RFID Sample Code component, the following programming samples are installed by default in the `RFID_EDGE_HOME/samples` directory, where `RFID_EDGE_HOME` represents the product installation directory. You can modify these sample applications and use them as a starting point for developing your own applications.

"Sample Java Applications" on page 6-1 provides procedures for setting up your development environment, as well as instructions for compiling, running, and working with some of the sample applications.

- `ImmediateProgramSample`

  An 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.

- `ImmediateSample`

  An example of how to use the ALE API to retrieve a list of Electronic Product Code (EPC) tags from a specified logical reader. The event cycle specification is read from an XML file, and the event cycle reports are printed as XML.

- `JMSSamples`

  Vendor-specific JMS examples for:

  – BEA

  – IBM

  – Sun

  – JBoss

  – TIBCO

- `NonXMLSample`

  An example of how to use the ALE API to retrieve a list of EPC tags from a specified tag reader. The `immediate()` method of the ALE client interface is used to perform the tag read operation.

- `PollingSample`

  An example of how to define an event cycle specification (ECSpec) and use it to poll() the Edge Server for tag updates.

- `ProgrammingSample`

  An example of how to define a programming cycle specification (PCSpec) by reading it from an XML-formatted file, how to administer an EPC cache in the Edge Server, and how to invoke tag programming operations.

- `SubscribeSample`

An example of how to define an event cycle specification (ECSpec) by reading it from an XML-formatted file, and how to set up a handler that subscribes to event cycle completion notifications.

- `Workflow`

  XML samples for use with the examples provided in the Configuring and Using Workflows section of the *RFID Workflow Reference* manual.

Reading and Writing Tags

The following sections provide an overview of the ALE API and describe how client applications use the ALE API to define requests for tag data:

- "Overview of the ALE API and ALE Operation" on page 2-1
- "BEA Implementation of the ALE API" on page 2-2
- "Benefits of the BEA Implementation" on page 2-3
- "Programming Methods" on page 2-4
- "Reading Tag Data" on page 2-4
- "Writing Tag Data" on page 2-9
- "Comparison of Event Cycles and Programming Cycles" on page 2-14

Note: In the sections of this programming guide that deal with reading tags, it is assumed that you have read The Application Level Events (ALE) Specification Version 1.0, which in addition to defining the API provides necessary background information. If you do not have a copy of the specification, you can find one at the EPCglobal site: http://www.epcglobalinc.org.

Overview of the ALE API and ALE Operation

Applications interact with an Edge Server through the ALE API. The application uses the ALE API to define tag reading requests and uses the BEA-provided ALEPC API to define tag writing requests. This section uses the term ALE API to refer to both APIs. The Javadoc for the ALE and ALEPC APIs is available online at the following URL:

`http://e-docs.bea.com/rfid/edge_server/docs20/javadocs/index.html`

ALE provides a high-level, declarative way to read and write RFID data, without requiring application programmers to interact directly with RFID readers or to perform any low-level real-time processing or scheduling operations. ALE logically occupies a position between application business logic and low-level RFID tag reads and tag writes, thereby providing a strong degree of insulation between the two.

Note: For information on configuring readers, see *Installing WebLogic RFID Edge Server* and *RFID Reader Reference*.

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

The basic concepts of ALE operation are straightforward:

- An application sends a request through the ALE interface in the form of an ECSpec to read tags, or in the form of a PCSpec to write tags.
- The ALE engine within the Edge Server processes the request, performs the requested actions (for example, filtering tag data during a read operation), and generates reports based on the conditions specified in the ECSpec or PCSpec.

A tag-reading request can be a one-time request that is satisfied synchronously (for example, “supply a list of the EPC codes that are currently stable in the field of readers at Dock Door 5”). Or the tag-reading request can be a standing request that generates asynchronous notifications when events of interest occur (“every ten minutes, how many new items have arrived”).

A tag writing request can also be satisfied synchronously (for example,“write the following tag now”). Or the tag-writing request can be a standing request that writes a tag and generates a report when an external event occurs (“when the case crosses the electric eye beam, write the tag”).

BEA Implementation of the ALE API

The BEA implementation of the ALE API provides:

- Support of the standard EPCGlobal Application Level Events (ALE) Specification

- API support for advanced tag features, including all features of the EPCglobal Class-1 Generation-2 UHF RFID Air Interface Specification (commonly called Gen2). This specification includes changes to the ALE API and the ALE schema and WSDL.
- Extensions to the standard ALE API, such as extra boundaries and Gen2 read support.
- ALEPC extensions for tag writing and advanced Gen2 support
- Reliable messaging using JMS(store and forward) or Web Services (WS-Reliable Messaging)
- Out-of-the-box RFID workflow samples
- Support of the standard Java or J2EE programming model for workflows
- RFID JMX support for workflow retrieval of edge and device telemetry data
- An RFID tag simulator for testing and development

For a full list of all the features in this release, see the Introduction and Roadmap section in the *Product Overview* manual.

Benefits of the BEA Implementation

The BEA implementation of the ALE API provides a number of unique benefits:

- **Rapid development and deployment:** Different teams can independently get new projects up and running quickly with a minimum amount of programming.
- **Flexibility:** 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 through a wide range of network transports and protocols.
- **Built-in support for multiple applications to share 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:** The ALE API supports a scalable number of readers per WebLogic RFID Edge Server, number of RFID Server instances per site, and number of sites in the overall application.

- **Standards leadership and support:** The BEA implementation of the ALE API supports and extends Version 1.0 of the ALE Specification.
- **WebLogic RFID Edge Server-specific extensions:** WebLogic RFID Edge Server extensions to the ALE API are explicitly noted throughout this document.

Programming Methods

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 WebLogic RFID Edge Server installation provides WSDL files that you can use for this purpose. The WSDL files are in your WebLogic RFID Edge Server installation directory at:

  ```
  /share/schemas/EPCglobal-ale-1_0.wsdl
  ```

  ```
  /share/schemas/ALEPCService.wsdl
  ```

- WebLogic RFID Edge Server provides a Remote Client library for the Java programming language, which Java programs can use to access a WebLogic RFID Edge Server through the ALE API.

Reading Tag Data

The following sections describe how tag data is read and reported:

- "Read Cycles and Event Cycles" on page 2-5
- "How Applications Interact with the Edge Server ALE Engine" on page 2-7
- "ECSpec Reports" on page 2-8

Applications define *event cycle specifications* (ECSpecs), which specify to the ALE engine what RFID data is of interest. An example of the content of an ECSpec is "send 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 WebLogic RFID Edge Server generates 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 Data Type" on page 4-7 and "ECReports Data Type" on page 4-23, respectively.

An application can define and subscribe to an ECSpec programmatically by using the ALE Remote Client to access the ALE API directly. Through the API, you can define an ECSpec and subscribe one or more destinations to an ECSpec for asynchronous delivery of ECReports. You can also use the API to request the delivery of an ECReports instance on demand, in a synchronous manner.

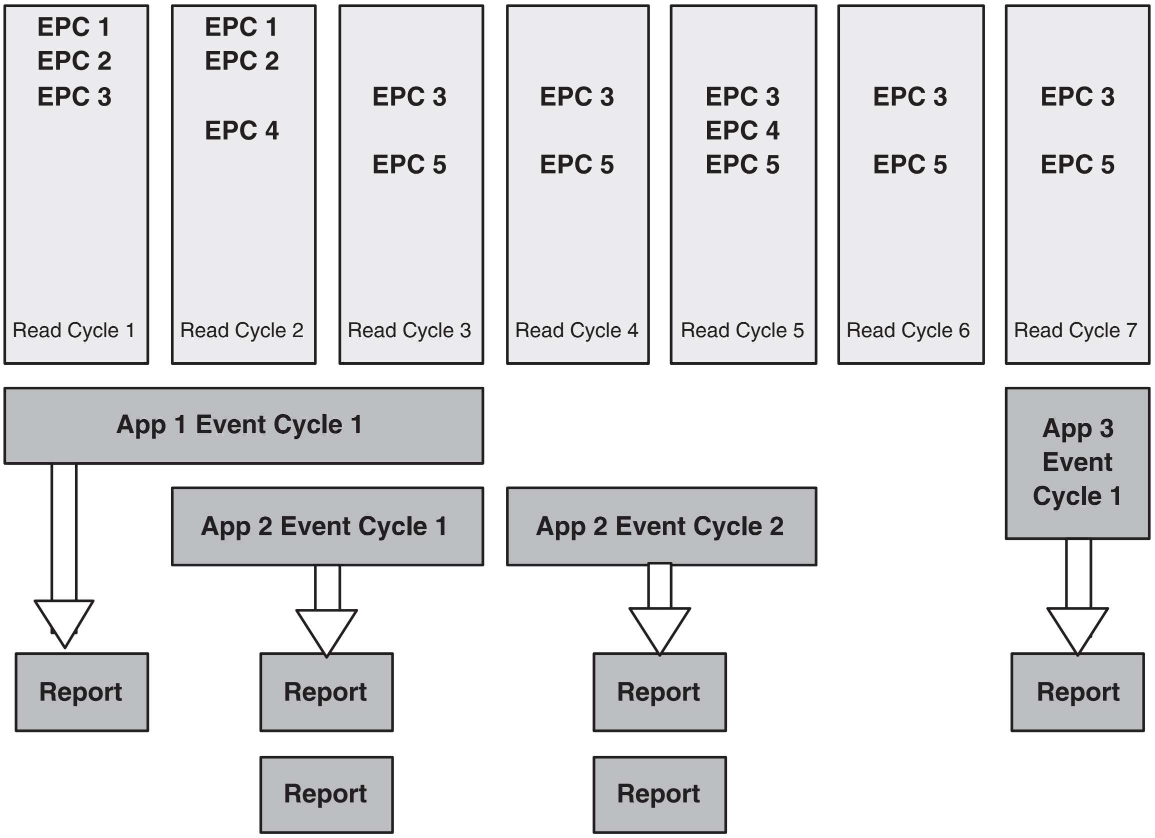
Read Cycles and Event Cycles

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 depends on many factors, including the position and motion of tags, presence of objects or people, and even the activity of other readers. Because of these factors, applications generally use the data accumulated from a number of RFID reads, and the ALE interface distinguishes between the rate at which readers scan for tags and 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 scan 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 tags seen in any read cycles by any of the readers.

At the completion of an event cycle, the ALE engine within the 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 can 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 can interact with a single Edge Server, many overlapping event cycles can be 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:

- Duration: Specify the event cycle to last a certain amount of time, independent of how many read cycles are involved.
- Number of read cycles: Specify the event cycle to last a fixed number of read cycles.
- Field stability: Specify the event cycle to complete when the set of tags read by a reader has been stable for a specified period of time (no new tags are seen during that period).
- External events: Specify the event cycle to begin or end when an external event occurs, such as a container passing an electric eye beam, or a human pressing a button.

Multiple end conditions can be specified, with the event cycle concluding when any of the conditions are met. For example, you can specify that an event cycle last 10 seconds OR the field is stable for at least 5 seconds.

Smoothing Read Cycles with Transient Filtering

WebLogic RFID Edge Server provides a transient filtering mechanism that can help produce smoother results when tags are not read reliably by readers. You can use the transient filter when you want 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 can 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). See *Installing WebLogic RFID Edge Server* for information on transient filtering.

How Applications Interact with the Edge Server ALE Engine

Applications interact with the RFID Edge Server ALE engine through event cycle specifications (ECSpecs) 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 also defines which locations (logical readers) are to be included; which 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. The WebLogic RFID Edge Server ALE engine can process large numbers of ECSpec instances from different applications simultaneously.

Three modes of interaction can occur between an application and an 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 by 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 continues 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. Subsequent reports are delivered through an asynchronous channel.

ECSpec 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 is listed once, even if it was detected in multiple read cycles on one reader or on multiple readers. Starting with this list of tags, each report is defined according to criteria applied in the following sequence:

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 the last event cycle for the same request, but which are no longer 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 might 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 4-17 includes a detailed description of filter options.
3. If there are no tags left after Step 2, should 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 might 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 4, "Reading Tags by 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 WebLogic RFID Edge Server supports tag writing, as well as the specific tag formats which are supported for each device, see the supported RFID readers section of the *RFID Reader Reference*.

Applications define *programming cycle specifications* (PCSpecs), 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 can 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.

The following sections provide information about various aspects of writing tag data:

- "Programming Cycles" on page 2-9
- "EPC Caches" on page 2-12
- "PCSpec Reports" on page 2-14

Programming Cycles

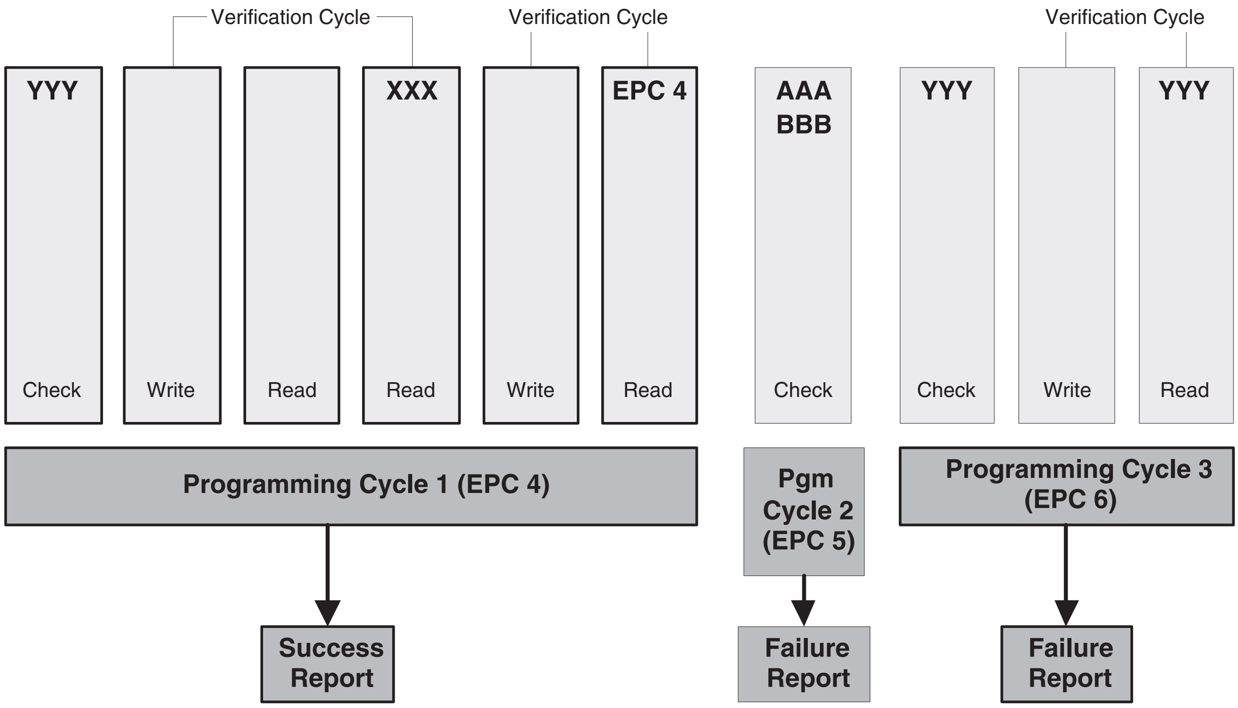
Tag writing services provided by the ALE API are organized around the notion of a programming cycle. A *programming cycle* is an interval of time during which a single tag is written and verified. Within a programming cycle, the reader attempts to ensure that there is a single tag in the field, write the tag, then read the tag to verify whether the write succeeded.

Like an event cycle, 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 (PCSpecs).

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 succeeds 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: A check operation is not needed before the second verification cycle, because the first verification cycle's read operations verify that a single tag is (still) in the field.

In Programming Cycle 2, the 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, 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 Cycles

Various reader manufacturers expose varying capabilities for tag writing. WebLogic RFID Edge Server maps between the definition of programming cycles, described in "Programming Cycles" on page 2-9, and the actual capabilities available on various types of readers.

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

In general, WebLogic RFID Edge Server presents a tag writing interface based on programming cycles as defined previously, 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

Programming cycles 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 that specifies the EPC value for each programming cycle. These use cases are handled through the use of EPC caches.

A PCSpec can be associated with an EPC cache, which is a collection of EPC values. Multiple PCSpec instances can share the same EPC cache. WebLogic RFID Edge Server 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 can 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 wants 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 might be supplied to the `defineEPCCache` operation, in which case the EPC cache is initialized in the replenished state. Alternatively, an application might 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.

Creating Tag Caches

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 the Reader Simulator and help you to learn how to create tag patterns. For information about the simulator, see Using the Reader Simulator.

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 might overlap or appear in non-ascending sequence. If an application wants 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.

PCSpec 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 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

WebLogic RFID Edge Server'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. Table 2-1summarizes the similarities and differences between event cycles and programming cycles.

Table 2-1 Comparing Event Cycles and Programming Cycles

Metric	Event Cycle	Programming Cycle
Direction	The flow of tag data is in one direction: from tag, through the 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 can 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 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 subscribe 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`.

CHAPTER 3

Asynchronous Notification Mechanisms

The following sections describe the asynchronous notification mechanisms WebLogic RFID Edge Server uses to deliver reports to client applications:

- "Overview of Asynchronous Notification Mechanisms" on page 3-1
- "Encoded XML Through HTTP POST" on page 3-2
- "Encoded XML Through TCP Socket" on page 3-2
- "Encoded XML in JMS Message" on page 3-3
- "XML Written to a File" on page 3-6
- "XML Displayed on the Edge Server Console" on page 3-7
- "XML Sent to a Workflow Module" on page 3-8

Overview of Asynchronous Notification Mechanisms

You can define specifications (ECSpec, PCSpec, and EPCCacheSpec) for your applications and later subscribe the application to asynchronous delivery of corresponding reports (ECReports, PCWriteReport, EPCCacheReport). WebLogic RFID Edge Server provides a number of ways to deliver asynchronous reports. A subscription 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.

In addition to out-of-the-box event delivery drivers that are applicable in a wide range of circumstances, the WebLogic RFID Edge Server provides an extensible mechanism for adding new delivery mechanisms. The following sections define the delivery mechanisms supported out-of-the-box by BEA, 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 4-34 (for `ECSpec` and `ECReports` objects)
- "XML Representations" on page 5-27 (for `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` objects)

Encoded XML Through HTTP POST

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

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

Table 3-1 Report Delivery Format: Encoded XML Through HTTP Post

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

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

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 indicates success; any other response code indicates failure.

Encoded XML Through TCP Socket

The Edge Server can 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
```

Table 3-2 Report Delivery Format: Encoded XML Through TCP Socket

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

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

Encoded XML in 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 [].

Table 3-3 Report Format: Encoded XML in JMS Message

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

`topic_name|` `queue_name`	Required	The name of the topic or queue to which the JMS notification driver sends its messages.
`queryParams`	Optional	Specify additional URI query parameters if necessary. To specify just one query parameter, append a string formatted as `?param1=value1` to the URI string. When you need to specify more than one parameter, append a string formatted as: `?param1=value1&param2=value2&param3=value3` Use the following query parameters: • `username` The user name that you specified when you created 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 acknowledgment 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`, 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. • 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 RFID Server also adds three additional properties to the `TextMessage`.

Table 3-4 Text Message Properties Added by RFID Edge Server

`specName`	The name of the `ECSpec` associated with this `ECReport`.
`savantID`	The RFID 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. For more information, see RFC 2396 at `http://www.ietf.org/rfc/rfc2396.txt`.

Examples of Report Delivery by Using XML with JMS

In the following example the Edge Server is instructed to send XML reports through JMS to a queue named `MyECReportSpecQueue`. The JNDI service is accessed using `iiop` at `jms.example.com` through 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
```

In the following example the Edge Server is instructed to send XML reports through JMS to a topic named `MyECReportSpecTopic`. The JNDI 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
```

The following example is more complex. This example incorporates JMS security as well as JNDI 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 service, and uses the username and password combination of `bob/PasswordForBob` to open a connection to the queue.

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

The following 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=Pas
swordForBob&secPrincipal=guest&jndi:java.naming.provider.url=iiop://jms.ex
ample.com:1099&jndi:java.naming.security.principal=guest&jndi:java.naming.
security.credentials=PasswordForGuest&jndi:java.naming.factory.initial=org
.jnp.interfaces.NamingContextFactory
```

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

```
jms:/queue/ConnectionFactory/MyECReportSpecQueue?username=bob&password=Pas
swordForBob&secPrincipal=guest&jndi:java.naming.provider.url=iiop://jms.ex
ample.com:1099&jndi:java.naming.security.principal=guest&jndi:java.naming.
security.credentials=PasswordForGuest&jndi:java.naming.factory.initial=org
.jnp.interfaces.NamingContextFactory&field1=value1&field2=value2
```

XML Written to a File

The Edge Server can 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 ensure 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 December 28, 2005 would be appended to the file

```
/mydir/myprefix-2005-12-28.xml
```

In all cases, the XML is as described in:

- "XML Representations" on page 4-34 (for `ECSpec` and `ECReports` objects)
- "XML Representations" on page 5-27 (for `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` objects)

XML Displayed on the Edge Server Console

The WebLogic RFID Edge Server can 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 might be useful to distinguish reports arising from different subscriptions. URI syntax restrictions prohibit the heading from being empty.

XML Sent to a Workflow Module

A workflow is a series of actions triggered by the observation of an EPC. The WebLogic RFID Edge Server can deliver ECReport data to workflow modules. Because the current release does not expose the workflow configuration APIs, use the Edge Server Administration Console for that purpose.

Reading Tags by Using the ALE API

The following sections describe the ALE API programming components that you use to read tags and include a formal, abstract specification of the ALE API. The external interface of the ALE API for reading tags is defined by the `ALE` class (See "ALE: Main Tag Reading Interface with UML Diagrams" on page 4-3). This interface uses complex data types that are documented in the sections starting at "ECSpec Data Type" on page 4-7. The ALE API is compliant with the EPCglobal ALE 1.0 specification.

- "Overview of the ALE API Implementation" on page 4-2
- "ALE: Main Tag Reading Interface with UML Diagrams" on page 4-3
- "Primary ALE API Data Types" on page 4-7
- "ECSpec Data Type" on page 4-7
- "ECReports Data Type" on page 4-23
- "Other ALE API Types: BEA Extensions" on page 4-31
- "XML Representations" on page 4-34
- "Using the ALE Tag Reading API from Java" on page 4-37
- "Gen2 Read Support" on page 4-38

Overview of the ALE API Implementation

One or more clients make method calls to the ALE interface. Each method call is a request, which causes the ALE engine to take an action and return results. Thus, methods of the ALE interface are synchronous.

The ALE interface also enables clients to subscribe to events that are delivered asynchronously, by using methods that take a URI as an argument. Such methods return results immediately, but subsequently the ALE engine within the Edge Server can asynchronously deliver information to the consumer denoted by the URI argument.

In the sections that follow, the API is described using Unified Modeling Language (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 and methods are marked as belonging to one of the following categories:

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

The ALE API is realized in several equivalent forms within the RFID Edge Server:

- 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 the ALE API 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. These differences are 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 makes clear the relationship to the UML. The UML-level descriptions in these sections are normative.

- For specifics of the Java binding, see the online Javadoc.
- For specifics of the WSDL binding, see the WSDL file in your installation directory under `./share/schemas`:

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

    Defines EPCglobal ALE schema; references BEA 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 schema extensions.

  Also see "XML Representations" on page 4-34.

ALE: Main Tag Reading Interface with UML Diagrams

`ALE` is the main Application Level Events (ALE) application programming 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
```

WebLogic RFID Edge Server 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.

An event cycle can be triggered in two ways:

- A standing `ECSpec` can be posted using the `define` method. Subsequently, one or more clients subscribe to that `ECSpec` using the `subscribe` method. The `ECSpec` generates 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. Each `ECSpec` instance is in one of three states: unrequested, requested, and active. An `ECSpec` is in the requested state it meets one or more of the following conditions:

- The ECSpec 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.
- The ECSpec 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.
- The ECSpec was defined using the `immediate` method, and the first event cycle has not yet been completed.

Once requested, an `ECSpec` is in the active state if reads are currently being accumulated into an event cycle based on the `ECSpec`. Standing `ECSpec` instances that are requested using `subscribe` can 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).

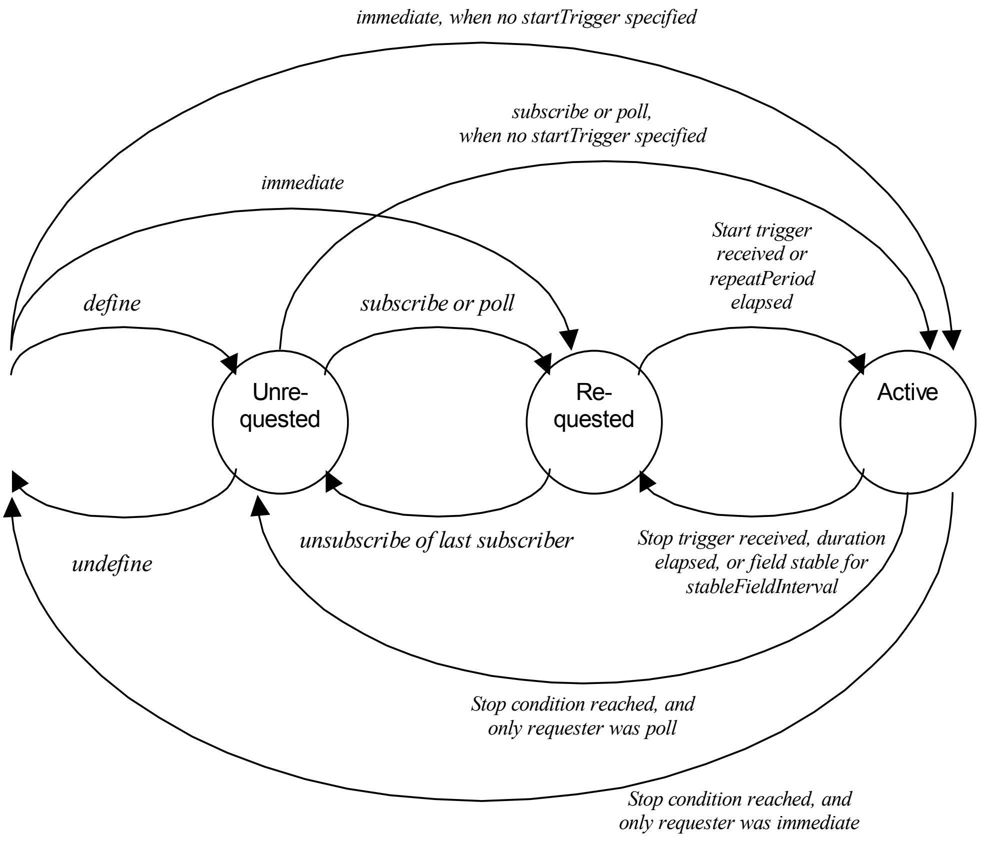
Two other methods are provided to manipulate ECSpec instances while preserving existing subscriptions:

- 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, because it might cause a gap in event cycle processing.

State Diagram

Figure 4-1 shows how the methods described in "ALE: Main Tag Reading Interface with UML Diagrams" on page 4-3 affect an ECSpec.

Figure 4-1 State Diagram


immediate, when no startTrigger specified
subscribe or poll,
when no startTrigger specified
immediate
Start trigger
received or
repeatPeriod
elapsed
define
subscribe or poll
Unre-
quested
Re-
quested
Active
undefine
unsubscribe of last subscriber
Stop trigger received, duration
elapsed, or field stable for
stableFieldInterval
Stop condition reached, and
only requester was poll
Stop condition reached, and
only requester was immediate

In addition, the two methods `getStandardVersion` and `getVendorVersion` return information about compliance with EPCglobal specifications:

- `getStandardVersion` returns a string that identifies which version of the EPCglobal ALE specification this implementation complies with. For this version of WebLogic RFID Edge Server, the method returns the string `1.0`.
- `getVendorVersion` returns a string that identifies the vendor extensions this implementation provides. For this version of WebLogic RFID Edge Server, this method returns the string `http://version.connecterra.com/ALE/1`.

Primary ALE API 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 Data Type" on page 4-7 and "ECReports Data Type" on page 4-23.

ECSpec Data Type

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

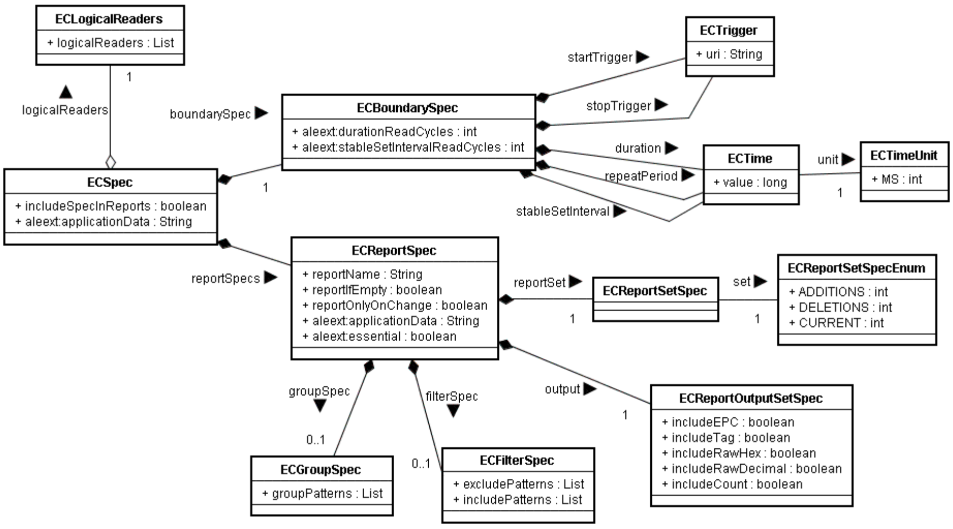
An `ECSpec` is a complex type that describes an event cycle and one or more reports that are to be generated from it. The following sections provide information about `ECSpecs`:

- "State Diagram" on page 4-6
- "ECBoundarySpec" on page 4-10
- "ECReportSpec" on page 4-13
- "ECReportSetSpec" on page 4-16
- "ECFilterSpec" on page 4-16
- "ECGroupSpec" on page 4-18
- "ECReportOutputSpec" on page 4-22

An `ECSpec` contains:

- A list of readers whose reader cycles are to be included in the event cycle. Each member of this list is either a single logical reader or the name of a composite reader.
- 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 that indicates whether or not to include the complete `ECSpec` as part of every `ECReports` instance generated by this `ECSpec`.
- An optional “application data” string, which is copied unmodified into every `ECReports` instance generated from this `ECSpec`.

Figure 4-2 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
```

WebLogic RFID Edge Server Extensions

```
applicationData : String

stableCount: int

---
```

Java Implementation Notes: 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` data members (`startTrigger`, `repeatPeriod`, and so on) directly. See the Javadoc and "ECBoundarySpec" on page 4-10.

Both an `ECSpec`'s associated `ECBoundarySpec` and `ECReportSpec` can contain an optional `<stableCount>` element. This element modifies the Stable Set Interval (SSI) stop condition on an Event Cycle by modifying the requirements that must be fulfilled before the SSI will cause an Event Cycle to terminate. Java access is provided by the following methods:

- `com.connecterra.ale.ECSpec.getStableCount() : int`
- `com.connecterra.ale.ECSpec.setStableCount(int) : void`

Note: It is an error to specify a `stableCount` element in a reportSpec that uses an `ECReportSetSpec` of `DELETIONS`.

If no `stableCount` elements are present, the stable set condition's semantics are identical to those in prior releases: the number of tags in the event cycle must be constant for the specified duration before the SSI will end the Event Cycle.

If one or more reports specify a `stableCount`, but the ECSpec has no top-level `stableCount`, the stable set interval is calculated only with respect to those reports that include the count. A report is considered stable if its included tag count has been stable for the entire stable set interval, and the included tag count is greater than or equal to the specified `stableCount`. The stable set interval ends the Event Cycle once all reports that contain a stableCount are stable.

If `stableCount` is specified at the ECSpec level, and no reports contain a `stableCount`, the count of tags in the event cycle must reach the specified value before the SSI condition will trigger the end of the event cycle.

If `stableCount` is present in reports and at the top level, both of the preceding conditions must be met before the SSI will cause the event cycle to end. Each report with an associated `stableCount` must be stable, and the number of tags in the event cycle must be stable and greater than or equal to the top-level `stableCount` before the SSI triggers.

If `stableCount` is specified but Stable Set Interval is specified (with either `stableSetInterval` or `stableSetIntervalReadCycles`), the rules apply as if the stable set interval is specified as 0, and ends the event cycle immediately when the stable count conditions are met. It is an error to specify an ECSpec with an implicit or explicit stable set interval of 0 unless at least one `stableCount` is specified nonzero.

ECBoundarySpec

An `ECBoundarySpec` specifies how the beginning and end of event cycles are determined, as described in the following sections:

- "ECBoundarySpec Implementation Notes" on page 4-12
- "ECTime" on page 4-12
- "ECTimeUnit" on page 4-13
- "ECTrigger" on page 4-13

EPCglobal ALE

```
startTrigger : ECTrigger

repeatPeriod : ECTime

stopTrigger : ECTrigger

duration : ECTime

stableSetInterval : ECTime
```

WebLogic RFID Edge Server 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 the time required to complete one read cycle for every reader that is included in the event cycle. 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: 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 preceding descriptions, an `ECSpec` presented through 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.

Note: URIs specify an event cycle's start or stop triggers. See *Triggers* 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 you use units of read cycles. If any reader experiences a failure during the event cycle, it will not complete its read cycles, and thus time as measured in read cycles might never reach the limits set for `duration` or `stableSetInterval`. For this reason, it is highly recommended that you 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 in an `ECBoundarySpec,` measured in physical time units. See "ECBoundarySpec" on page 4-10 and "ECBoundarySpec Implementation Notes" on page 4-12.

EPCglobal ALE

```
duration : long

unit : ECTimeUnit

---
```

ECTimeUnit

`ECTimeUnit` is an enumerated type denoting different units of physical time that can be used in an `ECBoundarySpec`. See "ECBoundarySpec" on page 4-10 and "ECBoundarySpec Implementation Notes" on page 4-12. `ECTimeUnit` currently supports only one time unit (milliseconds).

EPCglobal ALE

```
<<Enumerated Type>>

MS            // Milliseconds
```

ECTrigger

`ECTrigger` denotes a URI that specifies a start or stop trigger for an event cycle. See "ECBoundarySpec" on page 4-10 and "ECBoundarySpec Implementation Notes" on page 4-12.

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` contains one or more `ECReportSpec` instances.

EPCglobal ALE

`reportName : String`

`reportSet : ECReportSetSpec`

`filter : ECFilterSpec`

`group : ECGroupSpec`

`output : ECReportOutputSpec`

`reportIfEmpty : boolean`

`reportOnlyOnChange : boolean`

WebLogic RFID Edge Server Extensions

`essential : boolean`

`applicationData : String`

`stableCount : int`

`includedMemoryFields : List // of includedMemoryField : URI` (see "includedMemory" on page 4-38)

The `reportSet` parameter specifies the set of EPCs to be 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; that is, 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 from 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. In the event that more than one report is essential, all such reports must be present for an ECReports instance to be generated.

For example, in a shipment-receiving application there might 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 suppresses the generation of any other report, and no notification is 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), which is 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 `reportName` parameter enables a client to distinguish which `ECReport` instance that it receives corresponds to which `ECReportSpec` instance contained in the original `ECSpec`. This capability 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 Implementation Notes: `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 4-16.

- `get/setGroupSpec`

  See "ECGroupSpec" on page 4-18.

- `includeList`, `includeCount`

  See “ECReportOutputSpec” on page 4-22.

ECReportSetSpec

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

`ECReportSetSpec` is an enumerated type that specifies the set of EPCs 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 in “EPC Patterns” on page 4-17.)

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 ith pattern in the `includePatterns` list, and `Ei` denotes the set of EPCs matched by the ith 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 *EPC Tag Data Standard Version 1.1 rev 1.27*. 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 might not be present. Consult the EPCglobal *EPC Tag Data Standard* for details.) In an EPC pattern, each of those fields can 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 might 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" on page 4-18

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 *EPC Tag Data Standard*, 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.

Examples of Pattern URIs Used as Grouping Patterns

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 ensure 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. Assume `Pat_i` and `Pat_j` are two pattern URIs, written as a series of fields:

```
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 following table:

	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)
```

If each pattern is 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:

```
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 one 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 might 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.

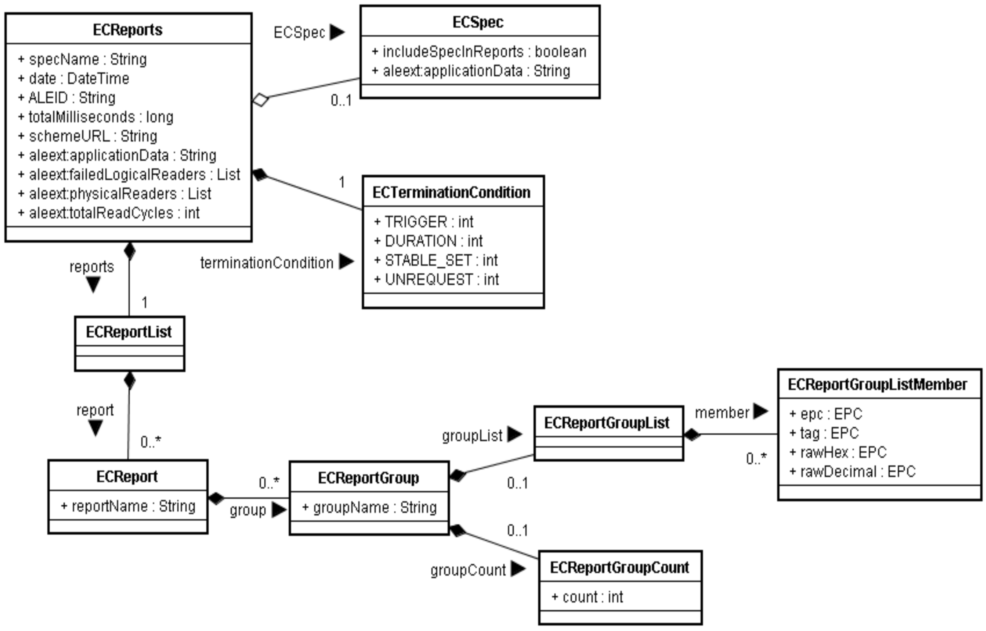
ECReports Data Type

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

The following sections provide information related to `ECReports`:

- "ECTerminationCondition" on page 4-25
- "ECReport" on page 4-26
- "ECReportGroup" on page 4-27
- "ECReportGroupList" on page 4-27
- "ECReportGroupListMember" on page 4-28
- "ECReportGroupCount" on page 4-30
- "ECReportGroupListMemberMemory" on page 4-30

Figure 4-3 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
```

WebLogic RFID Edge Server 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 most important part 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.

ECReports Header 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.

ECReports Header Field	Description
`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 4-10.
`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 for the XML schema for ALE API used in this version of WebLogic RFID Edge Server. This schema includes the BEA extensions to the EPCglobal ALE specification.
`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 4-10.

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 that is generated by an event cycle.

EPCglobal ALE
reportName: String
groups: List // List of ECReportGroup instances
WebLogic RFID Edge Server 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 4-18.

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 one 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. `ECReportGroupListMember` allows multiple EPC formats to be included and provides an extension point for adding per-EPC information to the list report.

EPCglobal ALE

`epc : URI`

`tag : URI`

`rawHex : URI`

`rawDecimal : URI`

WebLogic RFID Edge Server Extensions

`memoryItems` : List // See "ECReportGroupListMemberMemory" on page 4-30.

Each of the URI fields either contains a URI 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 *EPC Tag Data Standard* (`urn:epc:id:…`). A pure identity URI contains just the EPC, with no additional information such as tag type, filter bits, and so on. 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 *EPC Tag Data Standard* (`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 5, "Writing Tags by Using the ALEPC 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 *EPC Tag Data Standard* (`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 *EPC Tag Data Standard*. 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 Standard* 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.

ECReportGroupListMemberMemory

An `ECReportGroupListMemberMemory` provides access to the individual memory values returned in the ECReport from the Edge Server.

```
field : URI

value : URI
```

There are two URI forms for specifying a memory location on a tag:

- Absolute addressing (by bank, offset, and length)
- Symbolic name

When performing absolute addressing, the URI needs to include a selection for the memory bank, an offset within the memory bank, and a length of the memory extent, in bits.

```
urn:connecterra:tagmem:@bankid.length[.offset]
```

Where `bankid` is, in Gen2, one of the four values: `epc`, `tid`, `user`, or `reserved`. Length and offset are integer values in bits, specified as decimal. either decimal, or hexidecimal prefaced with `x`. The default offset value is 0.

When referring to a symbolic name, the URI simply includes the name.

```
urn:connecterra:tagmem:name
```

In the current implementation, the only valid value for `name` is `epc`. The value passed over the API for the contents of this memory is an EPC, and the Edge Server should perform whatever framing is necessary to read or write from the EPC memory bank on the tag. The use of the "epc" name also has implications for the formatting of the read memory in a return value. It's expected that the underlying implementation of this will simply include the EPC value returned by the inventory operation.

In either case we allow for future expansion by ignoring anything after the next colon in the URI. We may add explicit type information here in the future - for example, if the user knows that the first 96 bits of user memory represent a tag, we may allow a construction similar to

urn:connecterra:tagmem:@user.96:epc

Data is padded before being returned as a 64-, 96-, or 128-bit value. For example, 48 bits of data would be returned as a padded 64-bit value.

Other ALE API Types: BEA Extensions

The following sections define other types that are used in the WebLogic RFID Edge Server ALE API. These types are all BEA extensions to the EPCglobal specification.

- "ECSpecInfo (WebLogic RFID Edge Server Extension)" on page 4-31
- "ECSubscriptionInfo (WebLogic RFID Edge Server Extension)" on page 4-32
- "ECSubscriptionControls (WebLogic RFID Edge Server Extension)" on page 4-33

ECSpecInfo (WebLogic RFID Edge Server Extension)

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

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

WebLogic RFID Edge Server 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 might 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 (WebLogic RFID Edge Server 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 value is `0` if the most recent report was delivered successfully, `1` if the most recent report was not delivered but the preceding report was delivered, 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 (WebLogic RFID Edge Server 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. When a client wants to override these settings for a specific subscription, the client uses 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 can 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 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 asynchronous event cycle delivery mechanisms, as described in Chapter 3, "Asynchronous Notification Mechanisms." User applications can also use the XML forms 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 BEA 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 BEA schema extensions.

These files are located in your 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 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 WebLogic RFID Edge Server illustrate the use of the serializer and deserializer classes. See "Using XML Serializers and Deserializers from Java" on page 4-37 for more information.

The remainder of this section presents examples of `ECSpec` and `ECReports` as rendered into XML:

- "ECSpec - Example" on page 4-35
- "ECReports - Example" on page 4-35

These examples include additional line breaks and whitespace for the sake of readability. WebLogic RFID Edge Server permits (but does not require) this whitespace when reading XML; usually the server 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.

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 with UML Diagrams" on page 4-3 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 a WebLogic RFID 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 want to interact.

When using the `SOAPALEClient` class, you need to create instances of `ECSpec` and other types described in this chapter. 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 are in the product 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 by 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`.

Gen2 Read Support

In support of the UHF Generation 2 Air Interface Protocol, BEA provides the following properties:

- "includedMemory" on page 4-38
- "getMemoryItem" on page 4-39

includedMemory

ECReportSpec has a new `<includeMemory field="URI">` element, which can be specified multiple times. An `includeMemory` property is a list of URI objects, each of which corresponds to a tag memory section to read. Java access is provided by the following methods:

- `com.connecterra.ale.api.ECReportSpec.getIncludedMemoryFields() :List`
- `com.connecterra.ale.api.ECReportSpec.set(includedMemoryFields :List) : void`
- `com.connecterra.ale.api.ECReportSpec.addIncludedMemoryField(includedMemoryField :URI) : void`

When this element is specified, the associated ECReportGroupListMember in the report will have a `<memory field="URI">…</memory>` element whose contents are a URI representing the contents of the requested memory segment. When the field in the `includeMemory` element is an

EPC field, the contents are converted to a URI following the procedure as presented in the EPCglobal *EPC Tag Data Standard*. If the EPC must be returned raw, it is an EPC raw hex URI.

The value read is right-aligned; the LSB of the memory read is the LSB of the returned value, even when the width of the field is not a multiple of 8 bits.

getMemoryItem

The getMemoryItems() interface is an addition to the ECReportGroupListMember; the new interface provides access to the memory values returned in the ECReport from the Edge Server.

CHAPTER 5

Writing Tags by Using the ALEPC API

The following sections describe the ALEPC API programming components that you use to write tags and include a formal, abstract specification of the ALEPC API. The external interface is defined by the `ALEPC` interface (See "ALEPC: Main Tag Writing Interface with UML Diagrams" on page 5-3). This interface makes use of a number of complex data types that are documented in the sections starting at "PCSpec" on page 5-7. BEA extensions to the interface are described in "BEA Gen2 Write Support" on page 5-31.

- "Overview of the ALEPC API Implementation" on page 5-2
- "ALEPC: Main Tag Writing Interface with UML Diagrams" on page 5-3
- "PCSpec" on page 5-7
- "PCWriteReport" on page 5-17
- "EPCCacheSpec" on page 5-23
- "XML Representations" on page 5-27
- "Using the ALEPC Tag Writing API from Java" on page 5-30
- "BEA Gen2 Write Support" on page 5-31
- "Gen2 PCSpec Examples" on page 5-37

Overview of the ALEPC API Implementation

One or more clients 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 enables clients to subscribe to events that are delivered asynchronously. You implement this capability by invoking methods that take a URI as an argument. Such methods return immediately, but subsequently the ALE engine within the Edge Server can asynchronously deliver information to the consumer denoted by the URI argument.

In the sections that follow, the API is described by 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 ALEPC API is realized in several equivalent forms within WebLogic RFID Edge Server:

- There is a binding of the ALEPC API to Java, in which it takes the form of a collection of Java interface and class definitions.
- There is another binding of the ALEPC 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 ALEPC API has a slightly different structure and gives slightly different names to the different conceptual classes, data members, and methods defined in the UML within these sections. This variation 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 these sections should be considered normative.

- For specifics of the Java binding, see the online Javadoc.
- For specifics of the WSDL binding, see the WSDL file that is included in your 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 installation directory at:

  `share/schemas/ALEPC.xsd`

  See also "XML Representations" on page 5-27.

ALEPC: Main Tag Writing Interface with UML Diagrams

`ALEPC` is the main interface that you use to program tags. The term "tag programming" refers to the act of associating an EPC value with some physical entity, such as an RFID tag or a printed label.

The Java implementation package is `com.connecterra.alepc.api.` Also see "BEA Gen2 Write Support" on page 5-31.

```
---

getALEID() : String

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

poll(specName : String, parametermap : Map<String, String>) :
PCWriteReport

immediate(spec:PCSpec, epcVal:URI) : PCWriteReport

immediate(specName : String, parametermap : Map<String, String>) :
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
```

Table 5-1 Methods of the ALEPC Interface, in Alphabetical Order

Method	Description
`getALEID`	Return the value provided for `savantID` in the `edge.props` file.
`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. Use the overloaded method to provide a full map of parameters, which provide substitution values with the `PCSpec`.
listEPCCacheSpecNames	Return a list of the names of all EPC caches currently defined.

Method	**Description**
`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. Use the overloaded method to provide a full map of parameters, which provide substitution values with the PCSpec.
`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.
`undefineEPCCache`	Undefine an EPC cache.
`unsubscribe`	Unsubscribe a specified destination from receiving asynchronous delivery of reports from a specified programming cycle specification.

Method	Description
`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

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

Java implementation package: `com.connecterra.alepc.api.` See also "BEA Gen2 Write Support" on page 5-31.

The following sections provide information about a `PCSpec`:

- "PCSpecInfo" on page 5-9
- "PCSubscriptionControls" on page 5-10
- "PCSubscriptionInfo" on page 5-10
- "AccessSpec" on page 5-11

A `PCSpec` contains:

- A list of readers that should try to write the tag. Each member of this list is either a single logical reader or the name of a composite reader.
- 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" on page 2-12.

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

For a narrative description of programming cycles and their use of `PCSpec` instances, see "Programming Cycles" on page 2-9.

```
logicalReaders : List

readerParameters: Map

applicationData: String

cacheName: String

duration: long

startTrigger: URI

stopTrigger: URI

trials: int

accessSpecs: List

restrictSingleTag: boolean

---
```

Table 5-2 PCSpec Fields

Field	Description
`logicalReaders`	List of logical or composite reader names.
`readerParameters`	Maps parameter names to parameter values, and 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 *RFID Reader Reference* 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.

Field	Description
`trials`	Maximum number of EPC writing trials to run before failing a programming cycle.
`accessSpecs`	List of `AccessSpec` objects. See "AccessSpec" on page 5-11.
`restrictSingleTag`	Boolean that when true will cause a `PCSpec` to fail before a programming cycle if more than one tag is detected.

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

---
```

Table 5-3 PCSpecInfo Fields

Field	Description
`activationCount`	The number of times the programming cycle for the `PCSpec` has been activated since it was defined.
`cacheSize`	Number of entries that remain in the EPC cache associated with the `PCSpec`.
`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`.

Field	Description
`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. It is used by `PCSubscriptionInfo` (see "PCSubscriptionInfo" on page 5-10) and `ALEPC.subscribe(String, URI, PCSubscriptionControls)`. Also see "ALEPC: Main Tag Writing Interface with UML Diagrams" on page 5-3).

```
failureLimitCount : int

failureLimitInterval : long

---
```

Table 5-4 PCSubscriptionControls Fields

Field	Description
`failureLimitCount`	The maximum number of failed notification deliveries before a subscription is unsubscribed.
`failureLimitInterval`	The maximum interval of time a notification delivery can 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 5-10.
`lastSuccessTime`	The absolute time in milliseconds of the most recent successful notification.

AccessSpec

An `AccessSpec` contains an ordered list of `OpSpec` objects and a name. The list of `OpSpecs` represent the sequence of operations to be performed on each tag in field. The base `OpSpec` class is abstract; each subclass of `OpSpec` refers to a specific command.

The Java implementation package is `com.connecterra.alepc.api`. See also "BEA Gen2 Write Support" on page 5-31.

The following sections provide information related to an `AccessSpec`:

- "OpSpec" on page 5-12
- "DataSpec" on page 5-16

```
name : String
opSpecs : List
---
```

Field	Description
name	Name of AccessSpec object.
opSpecs	List of OpSpec objects. See "OpSpec" on page 5-12.

Java and XML Implementation Notes:

An `<accessSpec>` element (repeatable) in a `PCSpec` describes what commands to issue to each tag in a field, as they are "singulated." An `accessSpec` has a name in order to match it with an `accessReport` included in a `PCWriteReport`.

An `accessSpec` can contain a `<stopOnError>` subelement, which contains a Boolean value. If `stopOnError` is true, for any given tag, any error in an operation will halt processing for that tag. The inventory round then continues to the next tag in the field, and processing starts on that tag with the first operation. If omitted, `stopOnError` is assumed to be `false`.

A `PCSpec` that contains no `accessSpec` element acts as if it has a single implicit `accessSpec` element with a single write operation. If the `PCSpec` contains a `cacheName` element, the `writeOpSpec` uses that value for its `epcCache`. This is the only point at which the top-level `cacheName` element is used. Otherwise, the write specifies a `param` value of `epc`, which will be substituted using the data from the `poll` or `immediate` ALEPC API call.

If `restrictSingleTag` was specified in the `PCSpec`, explicitly or implicitly (through the lack of `accessSpecs`), the `PCWriteReport` contains the `epc` element which contains the EPC value of the single tag written. This element is omitted in the `PCWriteReport` of any `PCSpec` that does not restrict operations to a single tag, whether one more multiple tags were discovered in the field. Additionally, a `PCSpec` that contains no `accessSpec` element will use the older form of `PCWriteReport`, containing no `opReport`.

OpSpec

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

The base `OpSpec` class is abstract; each subclass of `OpSpec` refers to a specific command: `LockOpSpec`, `KillOpSpec`, `PasswordOpSpec`, `ReadOpSpec`, `WriteOpSpec`.

```
field : URI // Used by read and write

dataSpec: DataSpec // Used by kill, password, and write
```

```
mask : URI // Used by lock

value : URI // Used by lock

---
```

Table 5-5 OpSpec Fields

Field	Description
`field`	URI describing location on tag.
`dataSpec`	A `DataSpec` object. See "DataSpec" on page 5-16.
`opSpecs`	List of `OpSpec` objects.

Each `<accessSpec>` element contains an `<operations>` subelement, which is the list of `opSpecs` that describe the commands to issue to each tag ("operations"). These operations are `read`, `write`, `password`, `lock`, and `kill`.

There are two URI forms for specifying a memory location on a tag for a read or write operation:

- Absolute addressing (by bank, offset, and length)
- Symbolic name

For absolute addressing, the URI needs to include a selection for bank, an offset within the memory bank, and a length of the memory extent, in bits.

```
urn:connecterra:tagmem:@bankid.length[.offset]
```

Where `bankid` is, in Gen2, one of the four values: `epc`, `tid`, `user`, or `reserved`. Length and offset are integer values in bits, specified as decimal. either decimal, or hexidecimal prefaced with `x`. The default offset value is 0.

When referring to a symbolic name, the URI simply includes the name.

```
urn:connecterra:tagmem:name
```

In the current implementation, the only valid value for name is `epc`. The value passed over the API for the contents of this memory is an EPC, and the Edge Server should perform whatever framing is necessary to read or write from the EPC memory bank on the tag. The use of the "epc" name also has implications for the formatting of the read memory in a return value. It's expected that the underlying implementation of this will simply include the EPC value returned by the inventory operation.

In either case we allow for future expansion by ignoring anything after the next colon in the URI. We may add explicit type information here in the future - for example, if the user knows that the first 96 bits of user memory represent a tag, we may allow a construction similar to

urn:connecterra:tagmem:@user.96:epc

For a stack light device, the `bankid` has one value: `stacklight`. The memory bank is 20-bits wide, and each 4-bit segment corresponds to one light. For example, `urn.connecterra:tagmem:@stacklight.4.12` and value `urn:epc:raw:64.x9` will turn on the amber light.

The following code fragment demonstrates programmatic stack light control. The update value of `092F0` will turn off the White light (0), turn on the Blue light until next update (9), turn on the Green light for 10 seconds (2), do nothing to the Amber light (F), and turn off the Red light (0):

```
ALEPC alePCClient = new AxisALEPCClient(<alepcServiceURL>);
ALEPCFactory alePCFactory = AxisALEPCFactory.getInstance();
PCSpec pcSpec = alePCFactory.createPCSpec();
pcSpec.addLogicalReaderName("StackLight");
alePCClient.immediate(
             pcSpec, new URI("urn:connecterra:stacklight:update=092F0"));
```

For more information on stack light control, see Configuring and Controlling Stack Lights in the *RFID Reader Reference* manual.

A `<read field="...">` element is used to read from a segment of tag memory. The field attribute contains the URI to the tag memory location. The resulting `opReport` contains the status and the read memory value. Data is padded as a 64-, 96-, or 128-bit value before being returned. For example, 48 bits of data would be returned as a padded 64-bit value.

A `<write field=”...”>` element is used to write a segment of tag memory. The field attribute contains the URI to the segment tag memory location to write. The write operation contains one of three elements that can provide the data to be written to the specified memory. These elements, called data elements, are:

- `<literal>`*`dataURI`*`</literal>`: The dataURI is a raw or EPC URI that provides the data. The length of the data provided must match the length of the field.

  Note: For writing, there are two URIs: one is an address, the other is a value. The `<literal>` element includes the value URI. A value URI is also used to specify access and kill passwords, and lock bits. The address URI is also used to specify additional memory locations which should be read, in PCSpecs and ECSpecs.

- `<epcCache>`*`cacheName`*`</epcCache>`: Used to retrieve a value from the named EPC Cache.
- `<param>`*`parameterName`*`</param>`: Used to perform a lookup in the parameters provided the ALEPC `poll()` or `immediate()` call to find the value to write.

A `<password>` element attempts to transition the tag to secured mode. The password operation contains one of the data elements specified above in the `<write>` element description - this data is used as the password in the Gen2 "access" operation. It is an error to use an `epcCache` data element for this element.

A `<lock>` element contains a mask element and a value element, both of which should contain a raw URI that provides an integer value. These raw URIs should be 10-bits wide when operating with Gen2 tags. For any of the bits in the mask element that are set, the corresponding lock bit on the tag is set to the bit in value at that position.

- User memory is represented by the low 2 bits - 0x3
- TID is the next two - 0xC
- EPC is 0x30
- Access password is 0xC0
- Kill password is 0x300

A 10-bit value is provided for the "mask" and "value" of the `lock` command. In the mask, the bits that are set map to the bits in the value, to be applied to the state.

In each field, the high bit represents the "lock" value for that field, while the low bit represents the field's "permalock" value.

For the EPC, TID, and User memory fields:

lock	permalock	Description
0	0	Associated memory bank is writable.
0	1	Associated memory bank is writable.
1	0	Associated memory bank can only be written with a password.
1	1	Associated memory bank is permanently read-only.

For the password fields:

lock	permalock	Description
0	0	Associated password is readable and writable.
0	1	Associated password is permanently readable and writable.
1	0	Associated password can only be read or written with a password.
1	1	Associated password is permanently unreadable and inheritable.

Note the differences between the meanings for the bits in the two types of fields.

Lock operations are described in fuller detail in EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz-960 MHz, Version 1.0.9.

A `<kill>` element contains the kill password, which will be used to kill the tag. You specify this password using the same data subelement that the `<password>` element uses. It is an error to use an `epcCache` data element for this element.

For XML examples, see "Gen2 PCSpec Examples" on page 5-37.

DataSpec

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

The base `DataSpec` class is abstract. Public implementations are: `LiteralDataSpec`, `EPCCacheDataSpec`, and `ParamDataSpec`.

```
value : URI // Used by LiteralDataSpec

EPCCache : String // Used by EPCCacheDataSpec

paramName : String // Used by ParamDataSpec

---
```

Table 5-6 DataSpec Fields

Field	Description
`value`	An EPC URI or a raw URI.
`EPCCache`	Then name of an EPC Cache.
`paramName`	A parameter which is looked up in the map provided in an ALEPC `poll` or `immediate` call; the value associated with the parameter is used.

PCWriteReport

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

A `PCWriteReport` describes the tag writing operation of a programming cycle.

The following sections provide information related to a `PCWriteReport`:

- "AccessReport" on page 5-19
- "TagReport" on page 5-20
- "OpReport" on page 5-20
- "PCStatus" on page 5-21
- "PCTerminationCondition" on page 5-23

```
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

accessReport: AccessReport

---
```

Table 5-7 PCWriteReport Fields

Field	Description
`applicationData`	String that you set in the `PCSpec`. See "PCSpec" on page 5-7.
`cacheName`	The name of the EPC cache associated with this `PCSpec`.
`cacheSize`	Number of EPC cache entries remaining after 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.

Field	Description
`specName`	Name of the `PCSpec` that describes the just-completed programming cycle.
`status`	The status of the programming cycle. See "PCStatus" on page 5-21.
`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 5-23.
`totalMilliseconds`	The total time in milliseconds during which the programming cycle was active.
`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.
`accessReport`	A report based on an `AccessSpec`. See "AccessReport" on page 5-19 and "AccessSpec" on page 5-11.

AccessReport

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

An `AccessReport` contains the name of the associated `accessSpec`. Each `<accessReport>` element contains a list of `<tag epc="`*`epc`*`">` elements, one per tag detected in field.

```
name : String

tag : List

---
```

Table 5-8 AccessReport Fields

Field	Description
`name`	Name of AccessReport object. See "AccessReport" on page 5-19.
`tag`	List of TagReport objects. See "TagReport" on page 5-20.

TagReport

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

A `TagReport` contains the EPC of the tag, and an ordered list of `OpReport` objects. There is a one-to-one correspondence between each `OpReport` in the named `AccessReport` and each `OpSpec` in the corresponding `AccessSpec`, and the ordering is preserved.

```
EPC : URI

opReports : List

---
```

Table 5-9 TagReport Fields

Field	Description
`EPC`	The EPC of the tag. The tag `<epc>` value is formatted as a tag URI, as specified in the tag format data standard. If the tag is not a recognized type, it is formatted as a raw hex URI.
`opReports`	List of `OpReport` objects. See "OpReport" on page 5-20.

OpReport

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

An `OpReport` contains a status element that reports the operation was successful or provides failure information. For read and write operations, the `opReport` also contains memory operation results. These results include a field element that matches the memory bank URI specified in the original `opSpec`, and a value element that gives the value of the memory as a URI.

In the case of a write, the format of the value URI is exactly as provided as input - the literal write value, the parameter value, or the EPC URI provided by the cache. For a read, the value is formatted as described in the "includedMemory" on page 4-38

```
operationStatus : PCStatus

field : URI

value : URI

---
```

Table 5-10 OpReport Fields

Field	Description
`field`	See "OpSpec" on page 5-12.
`value`	See "OpSpec" on page 5-12.

PCStatus

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

`PCStatus` is an enumerated type that identifies the termination status of a programming cycle.

```
<<Enumerated Type>>

SUCCESSFUL

NONE_IN_FIELD

NOT_WRITTEN

VERIFY_ERROR

LOCKED

MULTIPLE_IN_FIELD

LOCKED
```

```
INCOMPATIBLE_TAG_TYPE

READ_ONLY

CACHE_EMPTY

READER_BUSY

READER_ERROR

ENGINE_ERROR
```

Table 5-11 PCStatus Values

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`	Reserved for future use.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 another programming cycle or by an event cycle.
`READER_ERROR`	One or more of the programming cycle's readers has a 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`

PCTerminationCondition is 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`

Table 5-12 PCTerminationCondition Values

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

An `EPCCacheSpec` describes a tag cache.

The following sections provide information related to an `EPCCacheSpec`:

- "EPCCacheReport" on page 5-24
- "EPCCacheSpecInfo" on page 5-25
- "EPCPatterns" on page 5-26

```
applicationData: string

includeCacheContent : boolean

threshold : long

---
```

Table 5-13 EPCCacheSpec Fields

Field	Description
`applicationData`	String that will be returned in `EPCCacheReport` instances. See "EPCCacheReport" on page 5-24.
`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 the number of remaining EPC values in a cache drops to (or below) the limit, an `EPCCacheReport` is issued to subscribers. Zero (0) means issue the `EPCCacheReport` when the EPC cache count drops to empty.

EPCCacheReport

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

An EPCCacheReport indicates that an EPC cache is low.

```
applicationData : String

cacheContent : EPCPatterns

cacheName : String
```

```
cacheSize : long

date : timestamp

savantID : String

threshold : long

---
```

Table 5-14 EPCCacheReport Fields

Field	Description
`applicationData`	String that you set in the `EPCCacheSpec`. See "EPCCacheSpec" on page 5-23.
`cacheContent`	Describes the remaining content of the EPC cache. See "EPCPatterns" on page 5-26.
`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

---
```

Table 5-15 EPCCacheSpecInfo

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 5-26.
`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

---
```

Table 5-16 EPCPatterns Field

Field	Description
`patterns`	EPC pattern URIs. The ordering of EPC patterns is significant. See "EPC Patterns" on page 4-17 and "EPC Caches" on page 2-12.

XML Representations

The focal points of the ALEPC tag writing interface from an application perspective are the `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` objects. The RFID Edge Server provides a standard means 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." The XML forms are also very useful to user applications as a means of interchange, and for persistent storage.

The XML forms are defined by the XSD schema. This schema is in the RFID Edge Server 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 can 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" on page 5-31 for more information.

The following sections present examples of `PCSpec`, `PCWriteReport`, `EPCCacheSpec`, and `EPCCacheReport` as rendered into XML. These examples include additional line breaks and whitespace for the sake of readability. RFID Edge Server permits (but does not require) this whitespace when reading XML; usually RFID Edge Server omits this whitespace when writing XML.

- "PCSpec - Example" on page 5-28
- "PCWriteReport - Example" on page 5-29
- "EPCCacheSpec - Example" on page 5-29
- "EPCCacheReport - Example" on page 5-30

For PCSpec examples that demonstrate Gen2 capabilities, see "Gen2 PCSpec Examples" on page 5-37.

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

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

Using the ALEPC Tag Writing API from Java

[[jtb: 29mar06: Per Chris, we do not mention AxisALECLient, it is deprecated and there is a SOAP replacment. But we do mention AxisALEPCClient because it is SOAP and there is no non-Axis replacement.]]

To use the ALEPC 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 with UML Diagrams" on page 5-3 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 WebLogic RFID 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 want 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 are located in your RFID Edge Server 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`.

BEA Gen2 Write Support

The following sections summarize BEA support for Gen2 command access:

- "Extended API Support" on page 5-32
- "Multiple Tags in Field" on page 5-36

- "Parameter Substitutions" on page 5-36

Extended API Support

In support of the UHF Generation 2 Air Interface Protocol, BEA provides extensions to the following classes:

- `com.connecterra.alepc.api.ALEPC`
  - `poll(specName :String, parameterMap :Map<String,String>) : PCWriteReport`
  - `immediate(specName :String, parameterMap :Map<String,String>) : PCWriteReport`

  See "ALEPC: Main Tag Writing Interface with UML Diagrams" on page 5-3.

- `com.connecterra.alepc.api.PCSpec`
  - `getAccessSpecs() : List`
  - `setAccessSpecs(accessSpecs :List) : void`
  - `addAccessSpec(accessSpec :AccessSpec) : void`
  - `isRestrictSingleTag() : boolean`
  - `setRestrictSingleTag(restrictSingleTag :boolean) : void`

  `accessSpecs` is a list of `AccessSpec` objects, defaulting to null. If `accessSpecs` is null, `restrictSingleTag` is ignored. Only when `accessSpecs` is set is the `restrictSingleTag` property meaningful: when `retrictSingleTag` is set `true`, the `<restrictSingleTag/>` element is added to the `PCSpec` and the Edge Server attempts to detect multiple tags in field (and fails the `PCSpec` if it detects that condition before programming). See"PCSpec" on page 5-7 and "AccessSpec" on page 5-11 for more information.

- `com.connecterra.alepc.api.AccessSpec`
  - `getName() : String`
  - `setName(name :String) : void`
  - `getOpSpecs() : List`
  - `setOpSpecs(opSpecs :List) : void`
  - `addOpSpec(opSpec :OpSpec) : void`

  See "AccessSpec" on page 5-11 and "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.WriteOpSpec`

  – `getField() : URI`

  – `getDataSpec() : DataSpec`

  `WriteOpSpec` specifies a write operation to a memory bank on the tag. It contains the memory bank URI specifying which tag field to write, and a `DataSpec` telling the Edge

  Server how to calculate the value to write to that memory. See "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.PasswordOpSpec`

  – `getDataSpec() : DataSpec`

  `PasswordOpSpec` represents an "access" command in the Gen2 tag protocol, which, if successful, moves the tag into "secured" state. The `PasswordOpSpec` contains a `DataSpec`, which tells the Edge Server how to compute the password for the tag. The provided `DataSpec` cannot be an `EPCCacheDataSpec`. See "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.KillOpSpec`

  – `getDataSpec() : DataSpec`

  `KillOpSpec` represents a "kill" command in the Gen2 tag protocol, which, if successful, permanently moves the tag into "killed" state. The `KillOpSpec` contains a `DataSpec`, which tells the Edge Server how to compute the kill password for the tag. The provided `DataSpec` cannot be an `EPCCacheDataSpec`. See "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.ReadOpSpec`

  – `getField() : URI`

  `ReadOpSpec` specifies a read operation from a memory bank on the tag. It contains the memory bank URI that specifies which tag memory to read. The associated `OpReport` in the `PCWriteReport` will contain the value read. See "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.LockOpSpec`

  – `getMask() : URI`

  – `getValue() : URI`

  `LockOpSpec` specifies a lock operation on the tag. It contains the mask and value for the lock operation. As the Gen2 specification describes, the lock operation attempts to modify the lock bits, using the bits of mask for the mask bits, and the bits of value for the "action" bits. See section 6.3.2.10.3.5 of the EPCglobal Gen2 spec. For Gen2 tags, the mask and value should be 10-bit raw EPC URIs. See "OpSpec" on page 5-12 for more information.

- `com.connecterra.alepc.api.DataSpec`

  There are three public implementations of `DataSpec`: `LiteralDataSpec`, `EPCCacheDataSpec`, and `ParamDataSpec`.

  – `LiteralDataSpec` holds a URI, either an EPC URI, or a raw URI.

  - `getValue() : URI`

  – `EPCCacheDataSpec` names an EPC Cache. The value provisioned by the cache is used.

  - `getEPCCache() : String`

  – `ParamDataSpec` names a parameter. The parameter is looked up in the map provided in the ALEPC `poll` or `immediate` call, and the associated value is used.

  - `getParamName() : String`

  The data provided must be compatible with the memory specified. For a specific memory bank range, the data provided by the `DataSpec` must be of the same size as the specified memory segment. The associated `OpReport` in the `PCWriteReport` will contain the value written. See "DataSpec" on page 5-16 for more information.

- `com.connecterra.alepc.api.ALEPCFactory`

  – `createAccessSpec(name :String) : AccessSpec`

  – `createPasswordOpSpec(opDataSpec :DataSpec) : PasswordOpSpec`

  – `createLiteralPasswordOpSpec(password :URI) : PasswordOpSpec`

  – `createParamPasswordOpSpec(paramName :String) : PasswordOpSpec`

  – `createKillOpSpec(opDataSpec :DataSpec) : KillOpSpec`

  – `createLiteralKillOpSpec(password :URI) : KillOpSpec`

  – `createParamKillOpSpec(paramName :String) : KillOpSpec`

  – `createReadOpSpec(field :URI) : ReadOpSpec`

  – `createWriteOpSpec(field :URI, opDataSpec :DataSpec) : WriteOpSpec`

  – `createLiteralWriteOpSpec(field :URI, value :URI) : WriteOpSpec`

  – `createEPCCacheWriteOpSpec(field :URI, epcCacheName :String) : WriteOpSpec`
  – `createParamWriteOpSpec(field :URI, paramName :String) : WriteOpSpec`
  – `createLockOpSpec(mask :URI, value :URI) : LockOpSpec`
  – `createLiteralDataSpec(value :URI) : LiteralDataSpec`
  – `createEPCCacheDataSpec(epcCacheName :String) : EPCCacheDataSpec`
  – `createParamDataSpec(paramName :String) : ParamDataSpec`

- `com.connecterra.alepc.api.PCWriteReport`
  – `getAccessReports() : List`

  An `AccessReport` contains a list of `TagReports`, each of which represents one tag detected in field. Each `TagReport` contains a list of `OpReports`, one for each operation performed on the tag. See "PCWriteReport" on page 5-17 and "AccessReport" on page 5-19 for more information.

- `com.connecterra.alepc.api.AccessReport`
  – `getName() : String`
  – `getTagReports() : List`

  An `AccessReport` has a name and a list of `TagReports`. The name matches the name of the AccessSpec that is associated with this access's `AccessReport`. There is one `TagReport` for each tag that was processed by the `AccessSpec`.

  The `accessReport` property of the `PCWriteReport` is `null` if the `PCWriteReport` returns no `accessReports`. See "AccessReport" on page 5-19 and "TagReport" on page 5-20 for more information.

- `com.connecterra.alepc.api.TagReport`
  – `getEPC() : URI`
  – `getOpReports() : List`

  The `TagReport` contains the EPC of the tag, and an ordered list of `OpReports`. There is a one-to-one correspondence between `OpReports` in the named `AccessReport` and `OpSpecs` in the corresponding `AccessSpec`, and the ordering is preserved. See "TagReport" on page 5-20 and "OpReport" on page 5-20 for more information.

- `com.connecterra.alepc.api.OpReport`
- `getOperationStatus() : PCStatus`

- `getField() : URI`
- `getValue() : URI`

  The `OpReport` contains the status of an `OpSpec` in the `AccessSpec`. The `OpReports` for `ReadOpSpec` and `WriteOpSpec` contain a field URI describing the memory that was operated on, and, if the operation was successful, the associated value from the memory. See "OpReport" on page 5-20 for more information.

Multiple Tags in Field

In Gen2, it becomes meaningful to perform work while multiple tags are in the reader field. For backward-compatibility, a `PCSpec` that is not identifiably Gen2 (a `PCSpec` that does not contain one or more `AccessSpecs`) will require that the field contain only a single tag, while a `PCSpec` that contains Gen2 elements will allow multiple tags by default.

A new `<restrictSingleTag/>` element has been added to the `PCSpec` to restrict a Gen2 `PCSpec` to return an error if there are multiple tags in field. This element is implicit in a `PCSpec` that contains no `AccessSpecs`. This restriction is on a best-effort basis, and does not represent a guarantee. The current release will only work on a single tag in field.

Parameter Substitutions

The ALEPC `poll()` and `immediate()` calls add `PCSpec` parameters, providing values for use within the processing of the Programming Cycle. The `poll` and `immediate` calls accept a parameter map: a map from parameter name to parameter value. The old-style ALEPC calls, which only take a single EPC value, will generate a parameter map from the parameter `epc` to the provided value.

The values in this parameter map can be used in three ways:

- Written to tag memory using the `<param>` value type
- Used as a password for `password` or `kill` commands (again using the `<param>` value type)
- Used to override reader parameters used by the driver

The `write`, `password`, and `kill` operations can take a `<param>` element for the data needed for the operation. The contents of that element are used as the key into the parameter map, and the associated parameter value is used as the data. For example, the `PCSpec` can specify the epc cache and a user memory value that indicates the manufacturer, and `poll()` can specify a date code.

Users of the `readerParameters` of the `PCSpec` have been modified to check these parameters for a match before searching the `readerParameters`.

Gen2 PCSpec Examples

The following PCSpec examples demonstrate XML encoding of some Gen2 features:

- "readEPCBank.xml" on page 5-37
- "readAbsolute.xml" on page 5-38
- "readPassword.xml" on page 5-38
- "writePassword.xml" on page 5-39
- "writePasswords.xml" on page 5-39
- "writeTagMemory.xml" on page 5-40
- "writeTagMemory.xml and PCWriteReport.xml" on page 5-40
- "kill.xml" on page 5-41
- "lock.xml" on page 5-42
- "stackLight.xml" on page 5-42

The following examples are complete: `readEPCBank.xml`, `writeTagMemory.xml` and `PCWriteReport`, and `stackLight.xml`. The remaining examples show only the `<accessSpec>` portion of the PCSpec. They use the same `<logicalReaders>` and `<boundarySpec>` elements as `readEPCBank.xml`.

readEPCBank.xml

For all tags in field, report the EPC value of that tag.

```
<PCSpec xmlns="http://schemas.connecterra.com/alepc">
  <applicationData>application specific data can go here</applicationData>
   <logicalReaders>
      <logicalReader>ConnecTerra1</logicalReader>
   </logicalReaders>
   <boundarySpec>
      <trials>1</trials>
      <duration>4000</duration>
   </boundarySpec>
   <accessSpec>
      <operations>
         <operation>
            <read field="urn:connecterra:tagmem:epc"/>
         </operation>
      </operations>
   </accessSpec>
</PCSpec>
```

readAbsolute.xml

For all tags in field, read the raw EPC bank including the CRC and control bits, and report that value.

```
   <accessSpec>
      <operations>
         <operation>
            <read field="urn:connecterra:tagmem:@epc.96"/>
         </operation>
      </operations>
   </accessSpec>
```

readPassword.xml

For all tags in field, use 0x0000AAAA as the (32-bit) access password and report the first 32 bits of the reserved bank, followed by the next 32 bits. In a Gen2 tag these areas of the tag are the kill password, and the access password, respectively.

```
   <accessSpec>
      <operations>
         <operation>
            <password>
               <literal>urn:epc:raw:96.x0000AAAA</literal>
            </password>
         </operation>
         <operation>
            <read field="urn:connecterra:tagmem:@reserved.32"/>
         </operation>
         <operation>
```

```
          <read field="urn:connecterra:tagmem:@reserved.32.32"/>
        </operation>
      </operations>
   </accessSpec>
```

writePassword.xml

For all tags in field, use 0x11111111 as the access password and write the EPC value specified there to the EPC bank.

```
<accessSpec>
   <operations>
     <operation>
       <password><literal>urn:epc:raw:64.x11111111</literal></password>
     </operation>
     <operation>
       <write field="urn:connecterra:tagmem:epc">
         <literal>urn:epc:tag:sgtin-96:1.0037000.123456.1</literal>
       </write>
     </operation>
   </operations>
</accessSpec>
```

writePasswords.xml

For all tags in field, write 0x00000001 as the kill password and write 0x00000002 as the access password.

```
<accessSpec>
   <operations>
      <operation>
         <!-- write x00000001 into the KILL password  -->
         <write field="urn:connecterra:tagmem:@reserved.32">
             <literal>urn:epc:raw:64.1</literal>
         </write>
      </operation>
      <operation>
         <!-- write x00000002 into the ACCESS password  -->
         <write field="urn:connecterra:tagmem:@reserved.32.32">
             <literal>urn:epc:raw:64.2</literal>
```

```
              </write>
           </operation>
        </operations>
     </accessSpec>
```

writeTagMemory.xml

For all tags in field, write that raw value to the full EPC bank, overwriting the CRC and control bits.

```
  <accessSpec>
   <operations>
    <operation>
     <write field="urn:connecterra:tagmem:@epc.128">
      <literal>urn:epc:raw:128.973572063680812240853843738361866</literal>
     </write>
    </operation>
   </operations>
  </accessSpec>
```

writeTagMemory.xml and PCWriteReport.xml

```
<PCSpec xmlns="http://schemas.connecterra.com/alepc">
    <applicationData>Write Tag Memory</applicationData>
    <logicalReaders>
        <logicalReader>ConnecTerra1</logicalReader>
    </logicalReaders>
    <boundarySpec>
        <trials>1</trials>
        <duration>4000</duration>
    </boundarySpec>
    <accessSpec>
        <operations>
            <operation>
                <write field="urn:connecterra:tagmem:epc">
                <literal>urn:epc:tag:sgtin-96:1.0037000.123456.1</literal>
                </write>
            </operation>
        </operations>
    </accessSpec>
</PCSpec>


<PCWriteReport date="2006-04-04T21:45:55.369Z" savantID="Warren"
 specName="write" totalMilliseconds="1141" totalTrials="0"
 xmlns="http://schemas.connecterra.com/alepc">
 <applicationData>Write Tag Memory</applicationData>
 <wasSuccessful>true</wasSuccessful>
 <status>SUCCESSFUL</status>
 <physicalReaders>
  <physicalReader>SimReadr</physicalReader>
 </physicalReaders>
 <failedLogicalReaders/>
 <cacheSize>0</cacheSize>
 <accessReport>
  <tag epc="urn:epc:tag:sgtin-96:1.0037000.123456.1">
   <opReport>
    <status>SUCCESSFUL</status>
    <field>urn:connecterra:tagmem:epc</field>
    <value>urn:epc:tag:sgtin-96:1.0037000.123456.1</value>
   </opReport>
  </tag>
 </accessReport>
</PCWriteReport>
```

kill.xml

For all tags in field, try to kill the tag with the provided (32-bit) kill password (0x22222222):

```
   <accessSpec>
      <operations>
         <operation>
            <kill>
               <literal>urn:epc:raw:64.572662306</literal>
            </kill>
         </operation>
      </operations>
   </accessSpec>
```

lock.xml

For all tags in field, try to perform the specified lock operation with the provided password (0x11111111):

```
<accessSpec>
   <operations>
      <operation>
         <password>
            <literal>urn:epc:raw:64.286331153</literal>
         </password>
      </operation>
      <operation>
         <lock>
            <mask>urn:epc:raw:64.3</mask>
            <value>urn:epc:raw:64.3</value>
         </lock>
      </operation>
   </operations>
</accessSpec>
```

stackLight.xml

Write tag memory with stack light information, turning on the Amber stack light indefinitely:

```
<?xml version="1.0" encoding="UTF-8"?>
<PCSpec xmlns="http://schemas.connecterra.com/alepc">
    <applicationData>
     Turn on Amber stack light indefinitely
     (starting from bit 12, 4 bits)
    </applicationData>
    <logicalReaders>
       <logicalReader>StackLight</logicalReader>
    </logicalReaders>
    <boundarySpec>
       <trials>1</trials>
       <duration>4000</duration>
    </boundarySpec>
    <accessSpec>
```

```
         <operations>
            <operation>
            <!-- write EPC memory -->
               <write field="urn:connecterra:tagmem:@stacklight.4.12">
                     <literal>urn:epc:raw:64.x9</literal>
               </write>
            </operation>
         </operations>
    </accessSpec>
</PCSpec>
```

For more information on stack light control, see Configuring and Controlling Stack Lights in the *RFID Reader Reference* manual.

Sample Java Applications

The following sections describe how to use the sample Java applications provided in your WebLogic RFID Edge Server installation. The sample applications illustrate the use of the Java language binding for the ALE interface. Unlike other parts of WebLogic RFID Edge Server, the sample applications are free for you to use and modify for your own purposes. You can use them as a starting point for developing your own applications.

- "Overview of Sample Java Applications" on page 6-2
- "Setting Up Your Development Environment" on page 6-2
- "Compiling and Running the Samples" on page 6-2
- "ImmediateSample: Getting Started Reading Tags" on page 6-3
- "ImmediateSample: Event Cycles and Reliability" on page 6-7
- "ImmediateSample: Reading from Different Readers" on page 6-8
- "SubscribeSample: Exploring Asynchronous Event Cycle Delivery" on page 6-8
- "ImmediateProgramSample: Writing Tags" on page 6-12
- "ProgrammingSample: Exploring Programming Cycles and EPC Caches" on page 6-16
- "JMS Samples" on page 6-21
- "Workflow Sample XML Files" on page 6-23

Overview of Sample Java Applications

Several samples are provided with WebLogic RFID Edge Server:

- `ImmediateSample` — Shows how to use the XML serializer and deserializer, and the `immediate` method. The sample program reads an `ECSpec` from an XML file, activates it for one event cycle using the `immediate` method, and displays the results in XML to the RFID Edge Server console.
- `SubscribeSample` — Shows how to use the `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.
- `Workflow` — Contains sample XML files that define workflows.

Setting Up Your Development Environment

To compile and run the sample applications, you need to install both the Java Development Kit (JDK^{TM}) 1.4 or later, and the WebLogic RFID Edge Server software.

For detailed information about system requirements, prerequisite software, and how to install WebLogic RFID Edge Server, see *Installing WebLogic RFID Edge Server*.

Compiling and Running the Samples

The instructions for running all samples are the same:

1. From the `RFID_EDGE_HOME/bin` subdirectory of your WebLogic RFID Edge Server installation, run these scripts in the following sequence:
   - `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 WebLogic RFID Edge Server installation).
3. Run the "build" script (`build.sh` or `build.bat` depending on your platform) from the command line. This script compiles the sample program.
4. Run the "run" script (`run.sh` or `run.bat` depending on your platform) from the command line. This script 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 6-8. The sample program connects to your Edge Server, carries out its task, and then exits.
   – `ImmediateProgramSample`, see "ImmediateProgramSample: Writing Tags" on page 6-12. 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 6-16. This sample shows you how to manipulate programming cycles and EPC caches.

For tutorial walk throughs of the samples, see:

- "ImmediateSample: Getting Started Reading Tags" on page 6-3
- "ImmediateSample: Event Cycles and Reliability" on page 6-7
- "ImmediateSample: Reading from Different Readers" on page 6-8
- "SubscribeSample: Exploring Asynchronous Event Cycle Delivery" on page 6-8
- "ImmediateProgramSample: Writing Tags" on page 6-12
- "ProgrammingSample: Exploring Programming Cycles and EPC Caches" on page 6-16

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 WebLogic RFID Edge Server. However, if you have an actual reader and tags, you can use them.

The sample program reads an `ECSpec` from the file `ECSpec.xml`, which is shown in the following example, without the comments that are in the real file. After you become familiar with the sample program, you are encouraged to experiment by changing this file to see what happens.

```
<?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" />
      </reportSpec>
    </reportSpecs>

    <aleext:applicationData>Application specific data can go here
    </aleext:applicationData>
</ale:ECSpec>
```

The logical reader is specified as `ConnecTerra1`, which is mapped to “Antenna 1” in the Reader Simulator by default. (If you installed your own reader, modify the `ECSpec` to refer to one of your logical readers.) The event cycle is exactly one read cycle — this is far smaller than you are likely to use in any real situation, but it will illustrate how the ALE interface works. The final section defines a report specification, which will return 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 6-2. 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 might 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 might want to run `ImmediateSample` again, as follows:

1. 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.

   Keep your eye on the `uhfAntenna1.readCycles` display; `uhfAntenna1` corresponds to the logical reader `ConnecTerra1` that was specified in the sample `ECSpec.xml`. Looking at this display shows 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 6-2.

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

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

ImmediateSample: Event Cycles and Reliability

The `ImmediateSample` application illustrates several 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, this example uses 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 6-3. In all likelihood, you will see fewer tags in the report than you did previously.

In the following procedure, you 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.

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%.
5. 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 4-10) can 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.
4. Run `ImmediateSample` again. In the report, you will 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.

The `SubscribeSample` works with XML files to define event cycle specifications, as does the `ImmediateSample`. 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 define event cycle(s) and add 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 `./bin` subdirectory of your WebLogic RFID Edge Server 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 `SubscribeSample` 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. The name `mycmdlinespec` is used here.

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, after the `ECSpec` is activated by a method such as `immediate`, `poll`, or, in this example, `subscribe`.

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:

   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. To experiment with a different kind of event delivery driver, first create a new directory in a file system that is accessible to the Edge Server. For example:

   ```
   mkdir /tmp/ale
   ```

   On the Windows platform, the equivalent command would be, for example:

   ```
   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 WebLogic RFID Edge Server extensions (which are listed as RFTagAware extensions in this release).

```
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 by 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 WebLogic RFID Edge Server supports tag writing. See the supported RFID readers section of the *RFID Reader Reference* manual for this information.

If you plan to run this sample with the simulator, see "Using ImmediateProgramSample with the Reader Simulator" on page 6-14.

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

1. From the `RFID_EDGE_HOME/bin` directory of your WebLogic RFID Edge Server 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.

   Important: If you are using the simulator, be sure to read the section "Using ImmediateProgramSample with the Reader Simulator" on page 6-14.

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 WebLogic RFID Edge Server 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 5-7). Part of the file is reproduced here — you can take a look at the complete file in the `samples` directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<PCSpec xmlns="http://schemas.connecterra.com/alepc">
    <applicationData>application specific data can go<
       here</applicationData>
    <logicalReaders>
```

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

    <boundarySpec>
        <trials>1</trials>
        <duration>4000</duration>
    </boundarySpec>
</PCSpec>
```

4. In a shell, type:

   ```
   ./run.sh epcValue
   ```

   where *`epcValue`* is the EPC to write to the tag, for example:

   ```
    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.)

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

   ```
   # ./run.sh urn:epc:tag:gid-64-i:1.4.10

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

   Received the following PCWriteReport:
   <PCWriteReport date="2006-03-02T13:45:50.199Z" savantID="EdgeServerID"
    specName="$immediate=2" total Milliseconds="800" totalTrials="0"
    xmlns="http://schemas.connecterra.com/alepc">
    <applicationData>application specific data can go
      here</applicationData>
    <wasSuccessful>true</wasSuccessful>
    <status>SUCCESSFUL</status>
    <physicalReaders>
     <physicalReader>SimReadr</physicalReader>
    </physicalReaders>
    <failedLogicalReaders/>
    <cacheSize>0</cacheSize>
    <epc>urn:epc:tag:gid-64-i:1.4.10</epc>
   </PCWriteReport>
   ```

   The console output includes a `PCWriteReport`, expressed in XML. (See “PCWriteReport” on page 5-17.) `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 (for example: `CACHE_EMPTY` or `READER_ERROR`; for the complete list of status codes, see "PCStatus" on page 5-21.)

Note: If you are using the Reader Simulator and see a `MULTIPLE_IN_FIELD` status message, you probably have more that one active tag in the simulator. Make sure that you only have one active tag, and that the value of the tag is the same as the one you specify when invoking `run.sh`. Refer to "Using ImmediateProgramSample with the Reader Simulator" on page 6-14 for information on configuring the reader simulator for use with this example.

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

The `<failedLogicalReaders>` element is empty, because no logical readers failed during this programming cycle. The `<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 5-23.)

The `<epc>` element contains the EPC value that was written to the tag, in this case:

```
urn:epc:tag:gid-64-i:1.4.10
```

jtb: 5apr06: Following (aside from mentioning Alien1, is not in PCWriteReport: 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 a short procedure on how to run the sample application using the Reader Simulator, followed by additional information on which tag types are recognized by the simulator:

1. Start the Reader Simulator and go to the Reader Simulator GUI.
2. Turn off (uncheck) all the tags for Antenna 2.

3. Turn off (uncheck) all but one of the tags for Antenna 1 (only one tag can be active in order for the sample program to run successfully).
4. Make the value of the remaining tag the same as the tag you will supply as an argument to `run.sh`, for example: `gid-64-i:1.4.10`.
5. Run the example using the tag defined in the previous step. For example:

   ```
   $ ./run.sh urn:epc:tag:gid-64-i:1.4.10
   ```

   You should see output similar to that shown in **"ImmediateProgramSample: Writing Tags" on page 6-12**.

The Reader Simulator provides support for writing the following tag types:

gid-64-i	sgln-64	sgtin-96	sgln-96	usdod-64
sgtin-64	giai-64	sscc-96	giai-96	usdod-96
sscc-64	grai-64	gid-96	grai-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, unpredictable results can 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 Standard*. 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 might 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
```

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 WebLogic RFID Edge Server supports tag writing. See the supported RFID readers section of the *RFID Reader Reference* for information.

   Assign this reader the logical reader name `ConnecTerra1`, 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 6-14 to understand the constraints of the simulator.

2. From the `./bin` subdirectory of your WebLogic RFID Edge Server 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 WebLogic RFID Edge Server 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 at the console window for the Edge Server (not the Administration Console). A low-cache report has been issued to the subscription you just created:

   ```
   <!-- test -->
   <EPCCacheReport date="2006-03-16T16:38:01.734Z" savantID="EdgeServerID"
   xmlns="http://schemas.connecterra.com/alepc">
    <cacheName>mycache</cacheName>
    <applicationData>application specific data can go 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-cache mycache urn:epc:pat:gid-64-i:1.5.[1-15]
   ```

   This stocks `mycache` with a range of 15 EPC values.

9. In a shell, type:

```
./run.sh cache-info mycache
```

The ProgrammingSample prints:

```
Programming Sample
info for EPC cache mycache: Received the following EPCCacheSpecInfo:
<EPCCacheSpecInfo xmlns="http://schemas.connecterra.com/alepc">
 <subscriberCount>1</subscriberCount>
 <pcSpecs/>
 <activationCount>0</activationCount>
 <replenishCount>1</replenishCount>
 <lastReplenished>2006-03-16T16:39:05.097Z</lastReplenished>
 <lastReported>2006-03-16T16:38:01.734Z</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, the name `myspec` is given to the `PCSpec` you 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:

    ```
    Programming Sample
    polling myspec...
      ...received response.

    Received the following PCWriteReport:

    <PCWriteReport date="2006-03-16T16:42:15.454Z" savantID="EdgeServerID"
    specName="myspec" totalMilliseconds="540" totalTrials="0"
    xmlns="http://schemas.connecterra.com/alepc">

     <applicationData>application specific data can go here
     </applicationData>
     <wasSuccessful>true</wasSuccessful>
     <status>SUCCESSFUL</status>
     <physicalReaders>
      <physicalReader>SimReadr</physicalReader>
     </physicalReaders>
     <failedLogicalReaders/>
     <cacheName>mycache</cacheName>
     <cacheSize>14</cacheSize>
     <epc>urn:epc:tag:gid-64-i:1.5.1</epc>
    </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:

<PCWriteReport date="2006-03-16T16:53:16.758Z"
savantID="EdgeServerID-bea" specName="myspec" totalMilliseconds="300"
totalTrials="0" xmlns="http://schemas.connecterra.com/alepc">

 <applicationData>application specific data can go here
 </applicationData>
 <wasSuccessful>false</wasSuccessful>
 <status>CACHE_EMPTY</status>
 <physicalReaders>
  <physicalReader>SimReadr</physicalReader>
 </physicalReaders>
 <failedLogicalReaders>
  <logicalReader>ConnecTerra1</logicalReader>
 </failedLogicalReaders>
 <cacheName>mycache</cacheName>
 <cacheSize>0</cacheSize>
 <failureInfo>EPC cache 'mycache' empty for PCSpec myspec</failureInfo>
 <terminationCondition>FAILURE</terminationCondition>
```

JMS Samples

The samples in this section show how to configure JMS options and naming properties on the WebLogic RFID 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.

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

1. Configure the JNDI provider and JMS server.

2. Configure the WebLogic RFID Edge Server:
   - Copy the sample `jms.options` and `naming.props` files from the `./samples/JMSSamples/BEA/etc` directory into the `./etc` directory of the installation directory.
   - Modify the `jms.options` file in the `./etc` directory of the installation directory with the appropriate paths for the specified environment variables.
   - Modify the `naming.props` file in the `./etc` directory of the installation directory with the appropriate values for the `java.naming.provider.url` property.
   - Modify the `edge.props` file in the `./etc` directory of the installation directory to set the following property to the fully qualified name for `naming.props`:

     ```
     com.connecterra.ale.notificationDriver.jms.default.namingPropertiesF
     ile
     ```
   - (*Only if you are running the Edge Server as a Windows Service*) Modify the `edge.wrapper.conf` file in the `./etc` directory of the 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 the following one to `edge.wrapper.conf`:

     ```
     wrapper.java.classpath.31=c:\InstallRoot\AppServer\lib\someJarFile.j
     ar
     ```

     You can also use a relative pathname. For example, assume the `.jar` files are in a folder called `AppServerLib`, located directly under the 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. Using the `startWeblogic` script provided by BEA, start the WebLogic Server from a console window.
5. From a Web browser, log in to the WebLogic Server Administration Console:

   ```
   http://localhost:7001/console
   ```

6. From the WebLogic Server Administration Console, deploy the `JMSTest.ear` file, which is located in the `RFID_EDGE_HOME/samples/JMSSamples/BEA/deploy` directory. (For information on deploying applications, see the WebLogic Server documentation at http://e-docs.bea.com.)

7. Start the RFID Edge Server.

8. Define an `ECSpec` to the Edge Server. For example, use the `SubscribeSample` to define `myECSpec`:

   ```
   run define myECSpec ECSpec.xml
   ```

9. Set a JMS subscriber to the defined `ECSpec`. For example, set a JMS subscriber for `myECSpec` reports with the following command (shown as two lines here, but entered as one line):

   ```
   run subscribe myECSpec
   jms:/queue/weblogic.jms.ConnectionFactory/jms%2FTestQ)
   ```

   Note: BEA provides `weblogic.jms.ConnectionFactory` and `weblogic.jms.XAConnectionFactory` as default connection factories.

10. View `JMSTest MDB` messages showing `ECReports` for the defined `ECSpec` in the console corresponding to the startWebLogic command.

Workflow Sample XML Files

The `RFID_EDGE_HOME/samples/Workflow` directory contains three sample directories whose contents are XML files that define workflows. The three sample directories are:

- `DirectionalPortal`
- `ObservePortal`
- `PalletPortal`

For information on how to define workflows and import files, see the *RFID Workflow Reference* manual.

Index

A

AccessReport 5-19
AccessSpec 5-11
ALE
    main tag reading interface 4-3
ALE API
    application interaction 2-7
    benefits 2-3
    event cycles 2-5
    introduction to specification 4-2
    overview 2-1
    read cycles 2-5
    reports 2-8
ALEPC
    main tag writing interface 5-3
Applications
    asynchronous (subscribe) mode 2-7
    immediate mode 2-7
    immediate with predefined request (poll) mode 2-7
    interaction with API 2-7
applications
    sample 6-1
Asynchronous notification mechanisms
    XML displayed on the Edge Server console 3-7
    XML via HTTP POST 3-2
    XML via JMS Message 3-3
    XML written to a file 3-6

D

Deserializers 4-37, 5-31

E

ECBoundarySpec 4-10
ECReport 4-26, 4-27, 4-28, 4-30
ECReportOutputSpec 4-22
ECReports 4-23
ECReportSetSpec 4-16
ECReportSpec 4-13
ECSpec 4-7
ECTerminationCondition 4-25
EPC cache 2-12
EPC Patterns 4-17
EPCCacheReport 5-24
EPCCacheSpec 5-23
EPCCacheSpecInfo 5-25
EPCPatterns 5-26
Event cycle specification
    example 2-4
Event cycles 2-5
examples 1-3
    ImmediateProgramSample 1-4
    ImmediateSample 1-4
    JMSSamples 1-4
    NonXMLSample 1-4
    PollingSample 1-4
    ProgrammingSample 1-4
    SubscribeSample 1-4
    Workflow 1-5

H

HTTP POST delivery 3-2

I

ImmediateProgramSample 1-4, 6-2, 6-12
ImmediateSample 1-4, 6-2, 6-3, 6-7, 6-8
includedMemory 4-38

J

Java 2-4
Java binding 4-3
  tag reading 4-37
  tag writing 5-30
  XML serializers and deserializers 4-37, 5-31
Java sample applications 6-1
JMS message delivery 3-3
JMS samples 6-21
JMSSamples 1-4

K

KillOpSpec 5-12

L

LockOpSpec 5-12

N

NonXMLSample 1-4

O

OpDataSpec 5-16
OpReport 5-20
OpSpec 5-12

P

PasswordOpSpec 5-12
PCSpec 5-7
PCSpecInfo 5-9
PCStatus 5-21
PCSubscriptionControls 5-10
PCSubscriptionInfo 5-10
PCTerminationCondition 5-23
PCWriteReport 5-17
PollingSample 1-4
Programming cycles 2-9
  definition 2-9
  how they differ from event cycles 2-14
  reader implementation of 2-11
  specification (PCSpec) 2-9
Programming languages supported 2-4
ProgrammingSample 1-4, 6-2, 6-16

R

Read cycles 2-5
Readers
  transient filtering 2-7
Reading tag data 2-4
ReadOpSpec 5-12
Reports
  tag reading 2-8
  tag writing 2-14
RFID Edge Server
  standards compliance 1-3

S

Sample applications
  ImmediateProgramSample 6-2, 6-12
  ImmediateSample 6-2, 6-3, 6-7, 6-8
  ProgrammingSample 6-2, 6-16
  SubscribeSample 6-2, 6-8
  Workflow 6-2
sample applications 6-1
samples 1-3
  ImmediateProgramSample 1-4
  ImmediateSample 1-4
  JMSSamples 1-4
  NonXMLSample 1-4
  PollingSample 1-4
  ProgrammingSample 1-4
  SubscribeSample 1-4

Workflow 1-5
Serializers 4-37, 5-31
stableCount 4-9
Standards compliance 1-3
SubscribeSample 1-4, 6-2, 6-8

T

Tag writing sample application 6-12, 6-16
TagReport 5-20
Transient filtering 2-7

W

Workflow example 1-5
workflow module
XML delivery of ECReport data 3-8
Workflow sample 6-2
WriteOpSpec 5-12
Writing tag data 2-9
WSDL binding 4-3

X

XML
deliver ECReport data to workflow module 3-8
ECReports example 4-35
ECSpec example 4-35
EPCCacheReport example 5-30
EPCCacheSpec example 5-29
PCSpec example 5-28
PCWriteReport example 5-29
representation of tag reading objects 4-34
representation of tag writing objects 5-27
schema for tag writing 5-30
XML displayed on the Edge Server console 3-7
XML via HTTP POST 3-2
XML via JMS Message 3-3
XML written to a file 3-6