# BEA WebLogic RFID Edge Server

## Driver Development

Version 3.0
May 2007

# Contents

## Introduction and Roadmap



## BEA WebLogic RFID Driver SDK Overview



## Sample Driver

# Index

CHAPTER 1

# Introduction and Roadmap

The following sections describe the scope and organization of this document:



## Document Scope and Audience

This guide describes how to develop device drivers for the WebLogic RFID Edge Server. The devices that may be developed include RFID devices, barcode readers, programmable logic controllers (PLCs), or stack lights.

The guide documents the SDK interfaces, classes, and a sample driver provided with the BEA WebLogic RFID Edge Server software.

## Guide to This Document

- This chapter, Introduction and Roadmap, describes the organization of this document.
- BEA WebLogic RFID Driver SDK Overview provides a general overview of the SDK and its components. The chapter introduces the concepts and terminology used throughout this guide.
- Sample Driver describes how to compile and deploy the sample driver. You can use this sample as a starting point for developing your own driver.

# Related Documentation

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

- *Installing WebLogic RFID Edge Server* describes how to install and configure BEA WebLogic RFID Edge Server Driver SDK.
- *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 BEA WebLogic RFID Edge Server Driver SDK.
- *ALE and ALEPC Javadoc* provides reference documentation for the ALE and ALEPC packages that are provided with WebLogic RFID Edge Server.
- *Driver and Edge Flow Javadoc* provides reference documentation for developing custom Edge Flows and Drivers.

CHAPTER 2

# BEA WebLogic RFID Driver SDK Overview

The following sections provide an overview of the BEA WebLogic RFID Edge Mobile SDK and describe how client applications use the SDK APIs to write mobile applications:



## Overview of the BEA WebLogic RFID Driver SDK

The Edge Server device framework supports device drivers that communicate with a wide variety of devices, operating in different modes. A given device may implement one or more modes, with different latency, throughput, and network usage trade-offs between them. Within the Edge Server, the goal is for a driver to support all modes implemented by the device, although not generally simultaneously. Drivers implemented in the field for a specific application may only support the specific modes required by that application.

The use cases include all of the following driver modes:

- Tag and/or label printer
- Polling mode returning nonsingulated tags
- Polling mode returning singulated tags
- Autonomous mode (always returns nonsingulated tags)
- Asynchronous mode returning nonsingulated tags

- Asynchronous mode returning singulated tags

Some polling devices may support reading from all device units at once, with sequencing implemented by the device. Other polling devices may only be able to read from a subset of device units at once, or only one. Some polling devices will be able to include in their results the device unit which observed a tag. Some may not be able to include this information.

Some autonomous and asynchronous devices will only support autonomous/asynchronous OpSpecs. Some will only support synchronous OpSpecs. Some will support a combination of these. It should be possible with autonomous devices to support autonomous tag operations across network failures, driver reconfigurations, and driver restarts without missing tags or clearing the buffer, if such behavior is enabled in configuration.

An Edge Server may connect to a device using one or more channel types:

- Edge Server-initiated byte stream
- Edge Server-initiated character stream
- Edge Server-initiated PLC connection
- Device-initiated byte stream
- Device-initiated character stream

For stream channel types, a given device may permit synchronous requests and/or asynchronous requests.

The framework handles connection failure and retry logic directly, so device drivers do not have to. In order for this logic to work, a number of framework abstract methods are declared to throw IOException. If an implementation of one of these methods sees a communications problem, it must throw an IOException so that the retry logic is invoked.

# Event-driven State Machine

To control resource usage, the Edge Server uses a system of thread pools and event queues. The system is configured to create a pool of threads which may be fixed in size, or grow as needed. Each device driver has an event queue, which also may be fixed in size or grow as needed. Each event on the queue is represented by an implementation of the event interface, which consists of a single `handle`() method.

The events on each queue are processed on the threads in the pool. Although there is no guarantee that all events will be processed on a single thread, it is guaranteed that no more than one thread will process a queue at a given time. This means only one event on a queue is processed at a time;

as a result, synchronization is unnecessary for data touched only by event handlers. Finally, there is an alarm clock which allows for an event to be added to a queue at a time in the future.

In order not to block the use of the threads in the pool, events should not block on long-lived operations. As an exception to this rule, it is acceptable to perform network I/O operations in event handlers if the read is expected to complete. While such requests can time out and block a thread for an extended period, writing a driver in a completely I/O event driven fashion is burdensome. If a read may block for an unbounded period of time (for example, while waiting for a tag to enter an antenna's field), then a listener should be registered with the channel to handle incoming asynchronous data.

If the driver blocks for a long time in an event handler, then the rule that only one event from a queue may be processed at a time will mean that all event processing on the queue will block until the long-lived operation finally completes. Because all operations, such as changing the list of DeviceUnitRequests, are performed via events, this will have a significant effect on the operation of the device.

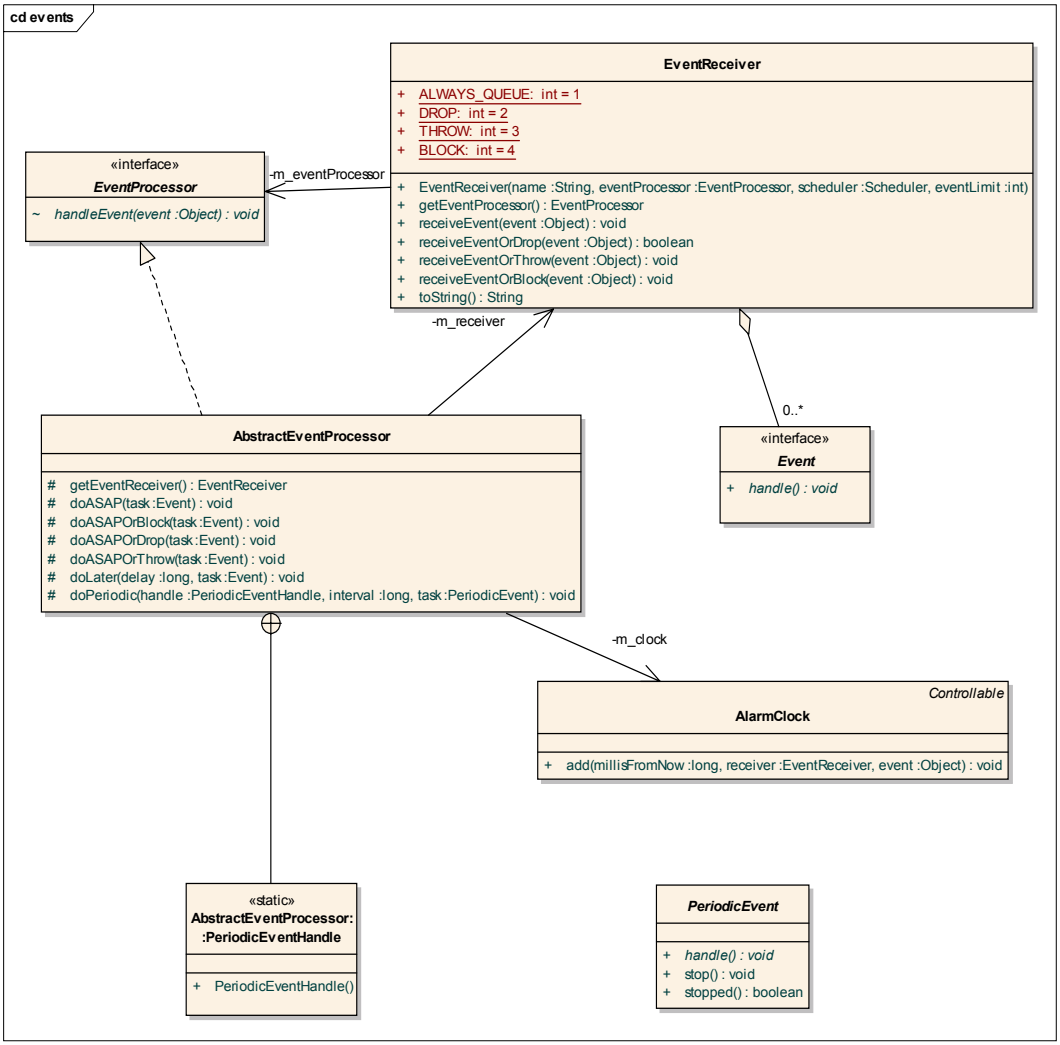
If a device defers work until a later time, it should use the alarm clock, not sleep. This event may be queued to occur as soon as all queued events complete, or it may be scheduled to occur at some time in the future (for example, a keepalive or timeout handler).

If an event scheduled for the future should be cancelled (such as a timeout which is no longer needed), then the driver should arrange for the event to be ignored. It may do this by using a serial number pattern. In this pattern, the driver stores a serial number which represents some iteration of an internal process. When that process is reset (for example, when a response to a request arrives, or times out), the serial number is incremented. Events which should not occur once the process completes should store the serial number when they are created, and check the current value when they are executed. If the stored serial number is not equal to the current one, then the event handler should return without doing anything.

Classes required:

- com.connecterra.util.event.Event
- com.connecterra.util.event.EventReceiver
- com.connecterra.util.event.EventProcessor
- com.connecterra.util.event.AbstractEventProcessor
- com.connecterra.util.event.AlarmClock
- com.connecterra.util.event.QueueFullException

- com.connecterra.util.event.PeriodicEvent




# Metadata and Initialization

Each driver must implement a static method called `getPluginMeta` which describes the configuration options required by the driver. This information is used by the Administration

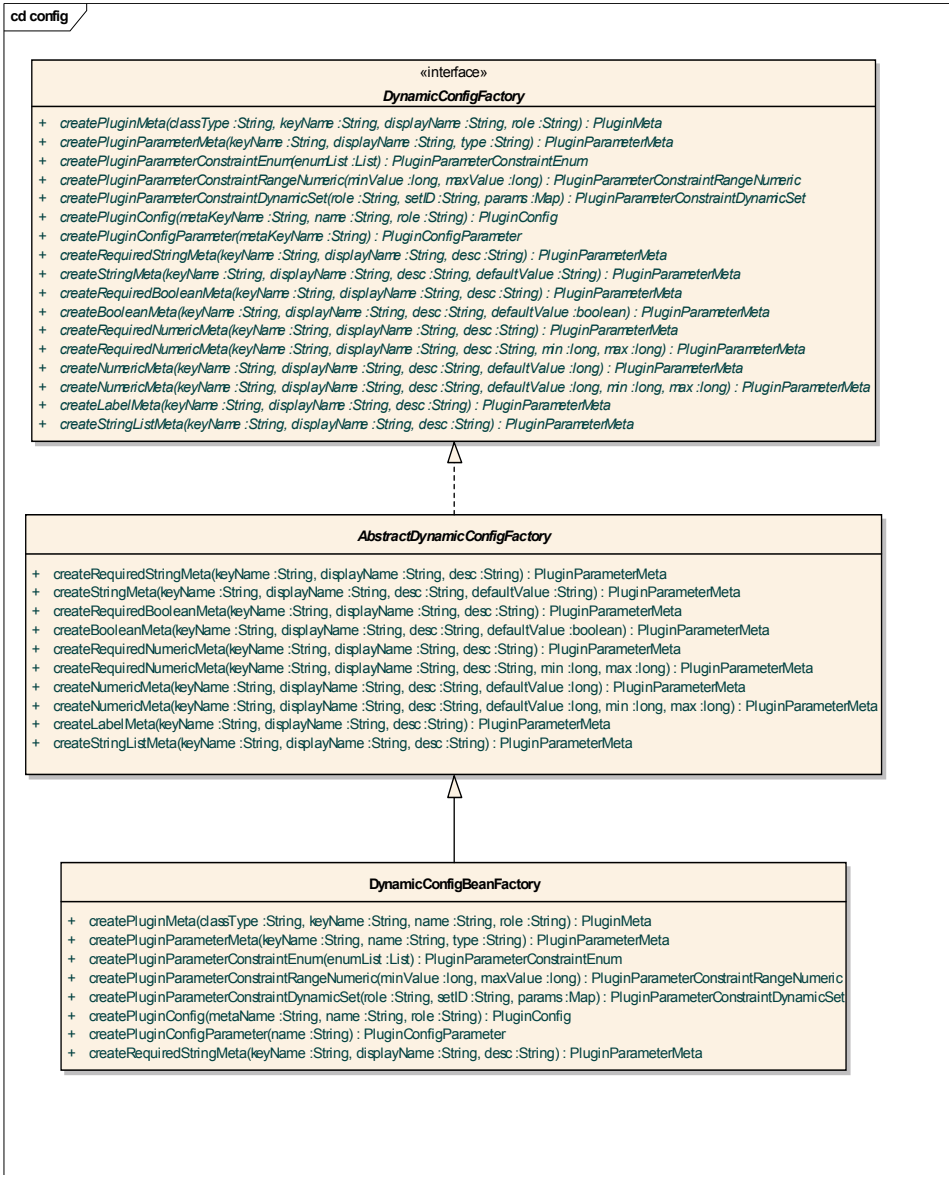
Console or other DynamicConfig clients to display configuration screens to the user, and to parse and validate configuration data received.

A module's `PluginMeta`, returned by the static method `getPluginMeta`(), contains the configuration information about the module.

Classes required:

- com.connecterra.ale.dynamicconfig.api.DynamicConfigFactory
- com.connecterra.ale.dynamicconfig.api.PluginMeta
- com.connecterra.ale.dynamicconfig.api.PluginParameterMeta
- com.connecterra.ale.dynamicconfig.api.PluginParameterConstraint
- com.connecterra.ale.dynamicconfig.api.PluginConfig
- com.connecterra.ale.dynamicconfig.api.PluginException
- com.connecterra.ale.dynamicconfig.api.DuplicateNameException
- com.connecterra.ale.dynamicconfig.api.PluginParameterConstraintDynamicSet
- com.connecterra.ale.dynamicconfig.api.PluginParameterConstraintEnum
- com.connecterra.ale.dynamicconfig.api.PluginParameterConstraintRangeNumeric
- com.connecterra.ale.dynamicconfig.api.NoSuchNameException
- com.connecterra.ale.dynamicconfig.api.PluginConfigParameter
- com.connecterra.ale.dynamicconfig.bean.AbstractDynamicConfigFactory
- com.connecterra.ale.dynamicconfig.bean.DynamicConfigBeanFactory
- com.connecterra.ale.reader.DeviceConfigurationException

When creating PluginMeta, all the output ports must be defined under the sub-configuration Module Outputs and all the input ports must be defined under sub-configuration Module Inputs. When driver instance is created, the AbstractPhysicalDevice.initialize() method is called. This method is passed a PluginConfig instance which contains the name and configuration information for the device. This method should perform any additional configuration verification necessary, and store the configuration values as needed. It must also create and register with the framework DeviceUnit objects corresponding to each device unit on the device, and DeviceChannel objects corresponding to each communication channel.


cd config
«interface»
DynamicConfigFactory
+ createPluginMeta(classType :String, keyName :String, displayName :String, role :String) : PluginMeta
+ createPluginParameterMeta(keyName :String, displayName :String, type :String) : PluginParameterMeta
+ createPluginParameterConstraintEnum(enumList :List) : PluginParameterConstraintEnum
+ createPluginParameterConstraintRangeNumeric(minValue :long, maxValue :long) : PluginParameterConstraintRangeNumeric
+ createPluginParameterConstraintDynamicSet(role :String, setID :String, params :Map) : PluginParameterConstraintDynamicSet
+ createPluginConfig(metaKeyName :String, name :String, role :String) : PluginConfig
+ createPluginConfigParameter(metaKeyName :String) : PluginConfigParameter
+ createRequiredStringMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createStringMeta(keyName :String, displayName :String, desc :String, defaultValue :String) : PluginParameterMeta
+ createRequiredBooleanMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createBooleanMeta(keyName :String, displayName :String, desc :String, defaultValue :boolean) : PluginParameterMeta
+ createRequiredNumericMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createRequiredNumericMeta(keyName :String, displayName :String, desc :String, min :long, max :long) : PluginParameterMeta
+ createNumericMeta(keyName :String, displayName :String, desc :String, defaultValue :long) : PluginParameterMeta
+ createNumericMeta(keyName :String, displayName :String, desc :String, defaultValue :long, min :long, max :long) : PluginParameterMeta
+ createLabelMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createStringListMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
AbstractDynamicConfigFactory
+ createRequiredStringMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createStringMeta(keyName :String, displayName :String, desc :String, defaultValue :String) : PluginParameterMeta
+ createRequiredBooleanMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createBooleanMeta(keyName :String, displayName :String, desc :String, defaultValue :boolean) : PluginParameterMeta
+ createRequiredNumericMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createRequiredNumericMeta(keyName :String, displayName :String, desc :String, min :long, max :long) : PluginParameterMeta
+ createNumericMeta(keyName :String, displayName :String, desc :String, defaultValue :long) : PluginParameterMeta
+ createNumericMeta(keyName :String, displayName :String, desc :String, defaultValue :long, min :long, max :long) : PluginParameterMeta
+ createLabelMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
+ createStringListMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta
DynamicConfigBeanFactory
+ createPluginMeta(classType :String, keyName :String, name :String, role :String) : PluginMeta
+ createPluginParameterMeta(keyName :String, name :String, type :String) : PluginParameterMeta
+ createPluginParameterConstraintEnum(enumList :List) : PluginParameterConstraintEnum
+ createPluginParameterConstraintRangeNumeric(minValue :long, maxValue :long) : PluginParameterConstraintRangeNumeric
+ createPluginParameterConstraintDynamicSet(role :String, setID :String, params :Map) : PluginParameterConstraintDynamicSet
+ createPluginConfig(metaName :String, name :String, role :String) : PluginConfig
+ createPluginConfigParameter(name :String) : PluginConfigParameter
+ createRequiredStringMeta(keyName :String, displayName :String, desc :String) : PluginParameterMeta

# Device Channels

The communications between the Edge Server and a device vary greatly. Sometimes the connection is a simple TCP socket which carries byte or character data. Sometimes the connection is not a single connection at all, but HTTP or SOAP which does not maintain any underlying network connection at all times (each HTTP request is often a separate TCP connection). Some connections are initiated by the device, not by the Edge Server. Some devices are implemented on top of an API that performs its own connection management.

In order to provide consistency, the framework provides the notion of a device channel. A driver may have more than one channel, each of which represents one potential means of communications between the Edge Server and the device. A channel may include one or more underlying network connections, and it may be initiated by the Edge Server or the device. A channel at any given time is either active or inactive. An active connection is one which is capable of carrying messages without any further setup on the part of the Edge Server.

For example, a TCP socket connection initiated by the Edge Server is a type of channel. This channel is active when the connection is alive, and inactive when the connection is down.

A TCP socket connection initiated by the device is also a type of channel. This channel is active whenever the TCP port is bound and listening, even if there is no socket open, because the device could open the a connection to the port and send a message at any time. When such a connection closes, the channel is still active.

If the channels are inactive, the framework will periodically attempt to reactivate the channels in order to resume operations on the device.

A set of commonly used channels is provided by the framework for all the modes supported. Drivers should use these channels when possible. When this is not possible (for example, when the channel is a wrapper around a library provided by the device vendor), then a new derived class of DeviceChannel should be created.

The initialize() method of the driver should create all the channels and add them to the framework. Note that this does not cause the channels to be activated, and the initialize() method must not attempt to activate the channels at this time. The framework handles channel activation and deactivation.

During operations, the various methods which can talk to the device are all declared to throw IOException. If the driver throws IOException, then the framework will execute a reset process which includes deactivating and reactivating each channel. (See the sequence diagrams for more details.)

If the channel is inactive due to a failure, it will only be reactivated again when the set of DeviceUnitRequests changes, or when the status poll interval expires for that device.

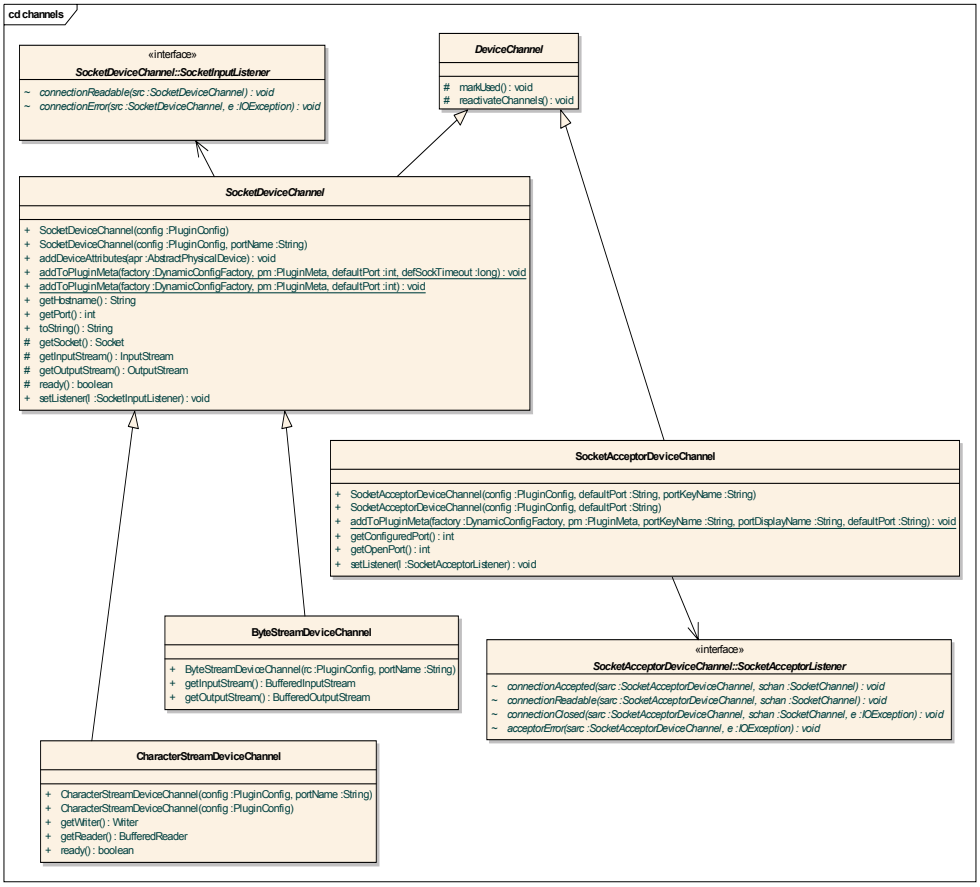
# Listeners

If a channel delivers data asynchronously to the Edge Server, the driver should make use of a channel with a listener. A listener is an interface which is registered with the channel, and contains callback methods when data is readable on a channel, or when a channel closes or gets an error. When a listener is registered, the channel will wait for data to be available. It will not read it, but instead queue an event on the driver's event queue which invokes the listener. This listener must then read a message from the device. A listener callback must only read as many messages from the device as are guaranteed to be present, or else the device's event queue will be blocked.

Such devices must also take care when performing request/response style operations, where a message is sent to the device, and a response is then read back. If a device is set up so that it might send an asynchronous message (containing tag data), then the response message might not be the expected response message at all, but instead an asynchronous message. The device can handle this in two ways. The device can implement a state machine where all responses, even to request/response style operations, are handled via the listener in an event-driven way. Or, the device can implement a request/response input algorithm which takes care to look at the message type information to determine if the message is a response to a request, or an asynchronous message. In the latter case, the driver must read and process the entire asynchronous message, then wait for the next message and try again (since the next message could also be either the response, or an asynchronous message).

Classes Required:

- com.connecterra.ale.reader.ChannelManager
- com.connecterra.ale.reader.DeviceChannel
- com.connecterra.ale.reader.ByteStreamDeviceChannel
- com.connecterra.ale.reader.CharacterStreamDeviceChannel
- com.connecterra.ale.reader.SocketAcceptorDeviceChannel




# Framework State Machine and Device Driver

After the device is initialized, the state machine for a given device driver is in one of four states:

- Channels inactive, with device unit requests
- Channels inactive, without device unit requests
- Channels active, with device unit requests
- Channels active, without device unit requests

The events that drive the transitions between states are:

- Device unit request set modified
- Session timer expired

  This method is called periodically (default is every fifteen seconds) to perform reconnect or status check operations on an otherwise idle session.
- Device poll

  If the driver is a polling mode driver, its processDeviceUnitRequests method is called periodically (configured as the default rate) whenever it has device unit requests.
- Channel input available
- Channel connection closed
- Channel input error
- Reactivate session

There are methods associated with each of the events above. A driver can override these methods, but it generally does not. Instead, the framework state machine converts the above events into calls to the following methods as appropriate (see the Javadocs and attached sequence diagram for details):

- AbstractPhysicalDevice.initializeSession(Requests)
- AbstractPhysicalDevice.updateRequests(Requests)
- AbstractPhysicalDevice.processDeviceUnitRequests(Set<AntennaRequest>)
- SocketInputListener.connectionReadable()
- SocketInputListener.connectionError([IOException])
- AbstractPhysicalDevice.sendStatusRequest()

Each driver must provide implementations which include business logic to configure and enable the device, operate on tags, and return tag data to the framework. The framework provides a number of support methods to simplify these tasks. There is an attached sequence diagram for each driver mode which describes best current practices behavior. Each of these methods is called from an event handler on the driver's event queue, so calls are guaranteed not to overlap.

# Device Unit Sequencing

If the device is a polling or autonomous device, AbstractPhysicalDevice.processDeviceUnitRequests() is called periodically to activate device units or read from the device's buffer. In either case, multiple DeviceUnitRequests for multiple device units may be simultaneously active if the device supports it.

For polling and autonomous modes, the framework provides support for sequencing DeviceUnitRequests. Specifically, given hints from the driver, the framework guarantee that all DeviceUnitRequests passed to a single call to processDeviceUnitRequests can be executed in a single command to the device (such device unit requests are called compatible). The DeviceCapabilities object contributes to this computation.

If the combined flag is set in the device capabilities, then the framework permits multiple device units to be created with the same logical reader name. Some devices will support this configuration directly, by internally treating both device units as one; in this case, it may not even be possible to determine what device unit observes a tag. Because they have the same logical reader name, they cannot be used separately. A DeviceUnitRequest on such a device will include all DeviceUnits with the same logical reader name when they are to be used.

The distinguished flag should be set in the device capabilities if the device can be instructed to read from multiple device units in a single command and the results identify for each tag which device unit or device units observed the tag.

The sequencer will invoke processDeviceUnitRequests with more than one DeviceUnitRequest if the requests are compatible. The default compatibility function checks if the following conditions are true:

- The OpSpec lists are the same
- The restrictSingleTag flags are the same
- The stopOnError flags are the same
- The device unit lists are the same, or the distinguished flag is set in the device capabilities

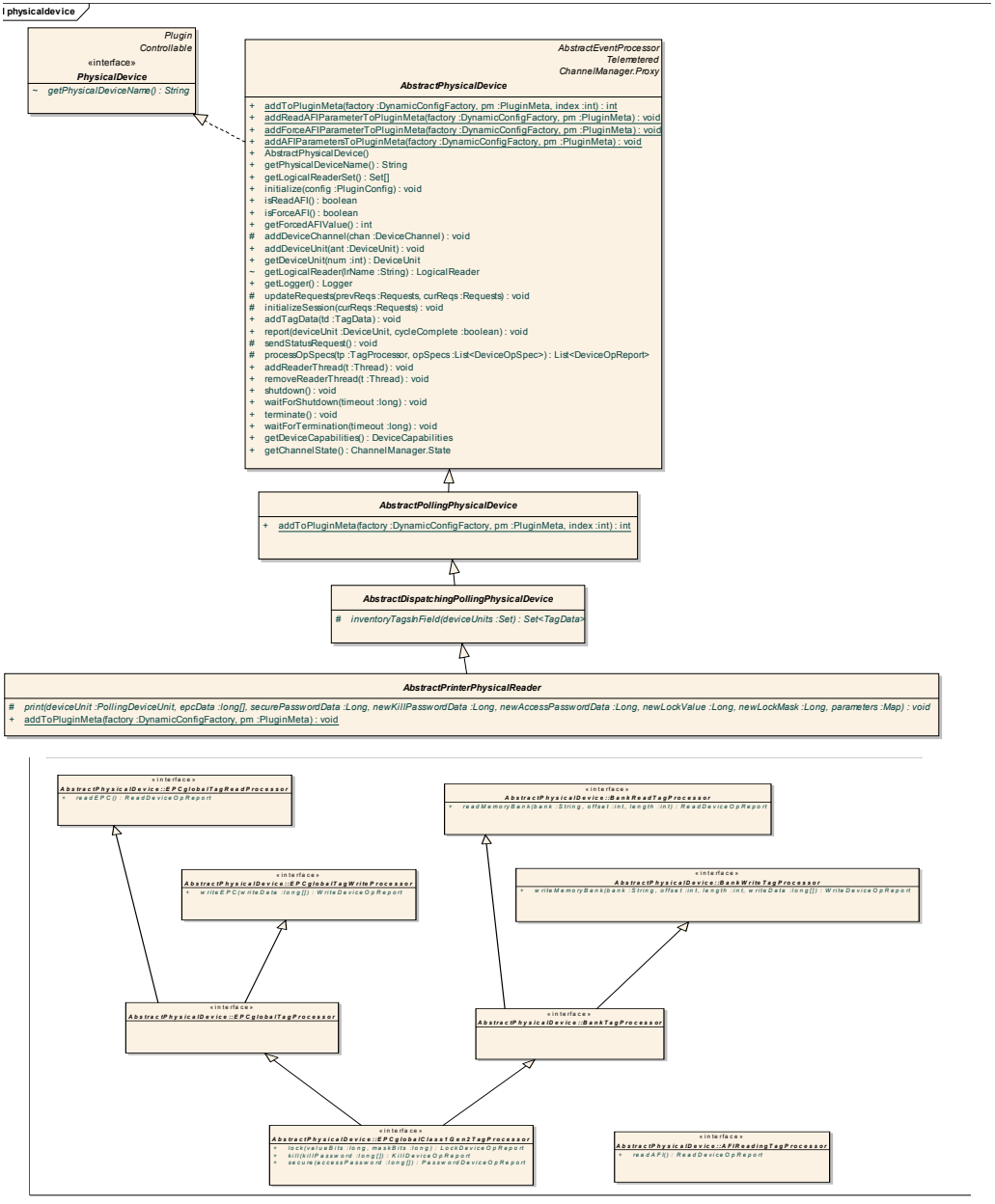
This definition is conservative. The method AbstractPollingPhysicalDevice.areDeviceUnitRequestsCompatible can be overridden to provide a driver-specific compatibility function if the default is not suitable.

For an autonomous mode driver to operate properly, the device must be capable of sequencing between all active device units by itself. If the device cannot do this, then a polling mode driver is more appropriate. If the buffer can be read only as a complete unit, then the compatibility

method must always return true. Otherwise, processDeviceUnitRequests will be called with disjoint subsets of active DeviceUnitRequests, and the buffer device will have no way to distribute the tag data to each DeviceUnitRequest correctly.

Classes required:

- com.connecterra.util.Controllable
- com.connecterra.ale.reader.PhysicalDevice
- com.connecterra.ale.reader.AbstractPhysicalDevice
- com.connecterra.ale.reader.AbstractPollingPhysicalDevice
- com.connecterra.ale.reader.AbstractDispatchingPollingPhysicalDevice
- com.connecterra.ale.reader.AbstractPrinterPhysicalReader
- com.connecterra.ale.reader.DeviceCapabilities
- com.connecterra.ale.reader.LogicalReader
- com.connecterra.ale.reader.DefaultLogicalReader
- com.connecterra.ale.reader.DeviceUnit
- com.connecterra.ale.reader.DeviceUnitCapabilities


physicaldevice
Plugin
Controllable
«interface»
PhysicalDevice
~ getPhysicalDeviceName() : String
AbstractEventProcessor
Telemetered
ChannelManager.Proxy
AbstractPhysicalDevice
+ addToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta, index :int) : int
+ addReadAFIParameterToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta) : void
+ addForceAFIParameterToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta) : void
+ addAFIParametersToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta) : void
+ AbstractPhysicalDevice()
+ getPhysicalDeviceName() : String
+ getLogicalReaderSet() : Set[]
+ initialize(config :PluginConfig) : void
+ isReadAFI() : boolean
+ isForceAFI() : boolean
+ getForcedAFIValue() : int
# addDeviceChannel(chan :DeviceChannel) : void
+ addDeviceUnit(ant :DeviceUnit) : void
+ getDeviceUnit(num :int) : DeviceUnit
~ getLogicalReader(lrName :String) : LogicalReader
+ getLogger() : Logger
# updateRequests(prevReqs :Requests, curReqs :Requests) : void
# initializeSession(curReqs :Requests) : void
+ addTagData(td :TagData) : void
+ report(deviceUnit :DeviceUnit, cycleComplete :boolean) : void
# sendStatusRequest() : void
# processOpSpecs(tp :TagProcessor, opSpecs :List<DeviceOpSpec>) : List<DeviceOpReport>
+ addReaderThread(t :Thread) : void
+ removeReaderThread(t :Thread) : void
+ shutdown() : void
+ waitForShutdown(timeout :long) : void
+ terminate() : void
+ waitForTermination(timeout :long) : void
+ getDeviceCapabilities() : DeviceCapabilities
+ getChannelState() : ChannelManager.State
AbstractPollingPhysicalDevice
+ addToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta, index :int) : int
AbstractDispatchingPollingPhysicalDevice
# inventoryTagsInField(deviceUnits :Set) : Set<TagData>
AbstractPrinterPhysicalReader
# print(deviceUnit :PollingDeviceUnit, epcData :long[], securePasswordData :Long, newKillPasswordData :Long, newAccessPasswordData :Long, newLockValue :Long, newLockMask :Long, parameters :Map) : void
+ addToPluginMeta(factory :DynamicConfigFactory, pm :PluginMeta) : void
«interface»
AbstractPhysicalDevice::EPCglobalTagReadProcessor
+ readEPC() : ReadDeviceOpReport
«interface»
AbstractPhysicalDevice::BankReadTagProcessor
+ readMemoryBank(bank :String, offset :int, length :int) : ReadDeviceOpReport
«interface»
AbstractPhysicalDevice::EPCglobalTagWriteProcessor
+ writeEPC(writeData :long[]) : WriteDeviceOpReport
«interface»
AbstractPhysicalDevice::BankWriteTagProcessor
+ writeMemoryBank(bank :String, offset :int, length :int, writeData :long[]) : WriteDeviceOpReport
«interface»
AbstractPhysicalDevice::EPCglobalTagProcessor
«interface»
AbstractPhysicalDevice::BankTagProcessor
«interface»
AbstractPhysicalDevice::EPCglobalClass1Gen2TagProcessor
+ lock(killBits :long, maskBits :long) : LockDeviceOpReport
+ kill(killPassword :long[]) : KillDeviceOpReport
+ secure(accessPassword :long[]) : PasswordDeviceOpReport
«interface»
AbstractPhysicalDevice::AFIReadingTagProcessor
+ readAFI() : ReadDeviceOpReport

# Device Unit Requests and OpSpecs

The work which a device should perform is described by a set of DeviceUnitRequests. A DeviceUnitRequest includes an unordered set of DeviceUnits, and an ordered list of OpSpecs. When a tag is observed by a DeviceUnit, the OpSpecs described in each DeviceUnitRequest which include the DeviceUnit must be executed on the tag, in order. In the simplest case, this DeviceUnitRequest may involve no more than reading the EPC of a tag.

This processing may be driven synchronously by the Edge Server on a device, or occur autonomously. Combinations may also be permitted. For example, the Impinj Mach1 protocol supports reading the EPC and one user memory field automatically, but must be explicitly commanded to perform other operations on the tag. (Nearly every device reports the EPC automatically for free, since the inventory includes this data.)

It should be noted that many devices support multiple modes of operation. It is not necessary for a device driver to support all the modes of operation of a device, or all the OpSpecs which the device can potentially implement. Various trade-offs in implementation time and complexity, network utilization, read latency, device performance, and user needs may affect what subset of functionality is implemented.

Because OpSpec processing is complex, the framework provides some help. The AbstractPhysicalDevice.processOpSpecs() method takes a TagProcessor and a list of OpSpecs, and invokes methods on the TagProcessor based on the OpSpecs in the list. Each method returns an OpReport specific to that method. These results are returned by processOpSpecs in a list whose elements correspond one-to-one with those in the list of OpSpecs. The TagProcessor class is abstract; the concrete class is defined by the device driver, and will often also implement one or more of a family of interfaces whose methods represent operations on tags which can be performed by that driver. The processOpSpecs() method will map the OpSpecs onto TagProcessor operations as efficiently as possible. If no interface is available to implement an operation, then an error OpReport is automatically generated by the class. In this way, the application can describe desired operations, the driver can describe the operations it can perform, and the framework can map between the two.

A TagProcessor may be used by updateRequests() to map a DeviceUnitRequest into the commands necessary to program a device to operate on tags automatically. Other possible uses for a TagProcessor are to operate on the tag directly when a poll event is received, or to help parse the response in an asynchronous driver. If the TagProcessor is used, then the DeviceOpSpec class and its derived classes should not need to be referenced directly.

In the future, the list of valid OpSpecs and TagProcessor interfaces will be extensible.

## Two-phase OpSpec processing

In the description above, each OpSpec results in a single call to an abstract method, which then performs the necessary work. This scheme works well when the relationship between OpSpecs and device operations is roughly one to one. However, it does not work if device operation is more coarse grained. Consider two use cases:

- An RFID label printer accepts a single command which causes a label to be printed and the tag in it to be programmed, and possibly assigned a kill password and locked. Because the printer implements all of this behavior as a single operation, making several calls to the printer is not possible. Instead, the driver must build up the operation internally, and issue it all at once.
- An asynchronous device can be set up to read from several banks, and report the set of results together back to the Edge Server. However, the descriptor which tells the device what parts of what banks to read is sent over as a single parameter, so it must be built up, and sent all at once.

The first use case is handled by the AbstractPrinterPhysicalReader class. This class is specific to RFID printer devices. It has a single abstract print() method which takes all the input from the DeviceUnitRequest and should be implemented by the device driver to print an program an RFID label.

The second use case is not supported by the framework.

Classes required:

- com.connecterra.ale.reader.DeviceUnitRequest
- com.connecterra.ale.uri.TagMemoryURI
- com.connecterra.ale.uri.EPCGen2Util
- com.connecterra.ale.reader.access.DeviceOpSpec
- com.connecterra.ale.reader.access.ReadEPCDeviceOpSpec
- com.connecterra.ale.reader.access.ReadBankDeviceOpSpec
- com.connecterra.ale.reader.access.ReadSymbolicDeviceOpSpec
- com.connecterra.ale.reader.access.WriteDeviceOpSpec
- com.connecterra.ale.reader.access.WriteLiteralDeviceOpSpec
- com.connecterra.ale.reader.access.WriteParamDeviceOpSpec

- com.connecterra.ale.reader.access.WriteEPCCacheDeviceOpSpec
- com.connecterra.ale.reader.access.LockDeviceOpSpec
- com.connecterra.ale.reader.access.PasswordDeviceOpSpec
- com.connecterra.ale.reader.access.KillDeviceOpSpec

## DeviceTagReports, Read Cycles, and OpReports

Once the device has done its work on the tags, it needs to report back what it has seen. When a tag is observed and has been operated upon due to a DeviceUnitRequest, a TagData object is passed to the framework using AbstractPhysicalDevice.addTagData(). The TagData object includes the identity of the tag (for EPCglobal tags, this is the EPC value and AFI, if any), the DeviceUnit which observed the tag, and a list of OpReports corresponding to the list of OpSpecs. An OpReport includes the data resulting from the OpSpec (for instance, the user memory data read, or the value of an EPC written from a cache), an error status code, and a string describing the error status in more detail.

Finally, after a set of tags is added, the framework method AbstractPhysicalDevice.report() must be called. This method actually sends the tags to the event cycle for a given DeviceUnit. This method has a boolean argument cycleComplete which must be set to true when a read cycle ends, even if no tags have been reported. This does the necessary bookkeeping for operations whose duration is based on the number of read cycles completed.

If the device has no notion of a read cycle, then the driver author may choose to pass in false for cycleComplete. Such a driver will never have a full read cycle, and so an event cycle on this reader with a duration in read cycles will never reach that termination condition. Or, it may choose to synthesize the cycle completion. This must be done with care, or else event cycles with a duration in read cycles and a report set spec of ADDITIONS or DELETIONS may return unpredictable results.

Classes required:

- com.connecterra.ale.reader.OpStatus
- com.connecterra.ale.reader.access.DeviceOpReport
- com.connecterra.ale.reader.access.ReadDeviceOpReport
- com.connecterra.ale.reader.access.WriteDeviceOpReport

- com.connecterra.ale.reader.access.LockDeviceOpReport
- com.connecterra.ale.reader.access.PasswordDeviceOpReport
- com.connecterra.ale.reader.access.KillDeviceOpReport
- com.connecterra.ale.reader.access.DeviceTagReport

## End of Life

When a device is undefined or redefined, the running instance is stopped by the Edge Server. Stopping the instance is done in two phases: shutdown and termination. When the shutdown() method is called to begin the first phase, the driver should begin the process of gracefully stopping device operations. For example, for an asynchronous mode driver, this might include removing the configuration of the host and port on the Edge Server where notifications are sent. This method should generally not block, using similar rules of thumb as event handlers. A second method, waitForShutdown() is then invoked by the Edge Server with a timeout. This method should perform any necessary wait() and join() methods to complete the shutdown.

The second phase, termination, should forcibly free any resources not already cleaned up by the shutdown process. This phase is implemented by the terminate() and waitForTermination() methods.

In both cases, the super class methods should be invoked in order to do framework level cleanup. The framework termination process will deactivate all the channels and terminate any listener threads which may have been started by the system.

# Implementation Roadmap

You have a new device. How do you work from there to a fully functioning driver? Here's a roadmap to make sure this process goes smoothly.

1. Familiarize yourself with the device.

   Read whatever developer documentation is available. Look at samples, maybe write some prototype code. Get a feel for how the device behaves.

2. Document use cases and requirements.

   Knowing this will help you make decisions later on in the development process. What are your tag throughput requirements? What are your latency requirements? What are you bandwidth requirements? Which is more important? (There may be more than one correct answer to these questions.) What OpSpecs do you need to support?

3. Determine what driver mode and channel types you need to use.

   This decision depends on the use cases and requirements above. If you have more than one set of requirements, you may need to support multiple driver modes. For example, a polling driver may reduce latency, but an asynchronous driver may reduce bandwidth. Some devices may support a choice of channel types which may or may not be tied to the choice of driver mode. If no single mode meets all the requirements, then the driver may support multiple modes, where the choice of what mode to use may be manual based on configuration options, or automatic based on the DeviceUnitRequests in place. Based on your choice, there are three base classes you should choose from:

   a. AbstractPrinterPhysicalReader - label printer devices with embedded EPCglobal RFID programmers for printing and programming on smart label stock.

   b. AbstractDispatchingPollingPhysicalReader - drivers which require some periodic synchronous communications with the device. For example, devices which read synchronously based on periodic requests from the Edge Server (polling mode drivers), or devices which read autonomously into a buffer which is then queried periodically by the Edge Server (autonomous mode drivers).

   c. AbstractPhysicalDevice - devices which have no need for periodic synchronous communications with the device. For example, devices which perform read operation on their own, and asynchronously deliver data to the Edge Server without the Edge Server having to request it first.

4. Determine what configuration options your driver needs to take.

   At a minimum, you will need to configure DeviceChannels and logical reader names for each DeviceUnit. Beyond that, you may choose to expose parameters which allow the deployer or developer to choose between different behaviors in the driver, or to set configuration options such as antenna gains on the device itself.

5. Implement your driver.

   Provide implementations for all necessary abstract methods and other base class methods. Create concrete TagProcessor and TagData objects as necessary to implement the driver's behavior.

6. Test your driver.

   a. Using the Admin Console, create ECSpecs that exercise the various functions of the device. For example:

      - Create an ECSpec and do Activate Once.
      - Create an ECSpec and subscribe to it.

- Create and read from an ECSpec which includes a user memory field.
- Use the Read Tags test provided by the Admin Console.
- Create an ECSpec which refers to multiple DeviceUnits.
- Create multiple ECSpecs which refer to disjoint DeviceUnits.
- Create multiple ECSpecs which refer to overlapping DeviceUnits.

b. Using ImmediateProgramSample:

- Program a tag.
- Lock a tag. Confirm that it cannot be written without a password. Confirm that it can be written with a password.
- Kill a tag. Confirm that it is dead.

7. Test for channel failure and recovery, with and without ECSpecs running:

- Redefine your reader in the admin console.
- Turn the reader off for a while. Observe how the system behaves (make sure it reports a problem). Turn it back on again.
- Turn the reader off and back on immediately.
- Unplug the network from the reader for a while. Observe how the system behaves. Turn it back on again.
- Unplug the network briefly and plug it back in. (This might not report any errors at all.)

Not all of the above tests will be meaningful for every device, and some devices should be tested in additional ways.

# Sample Driver

This section describes how to compile and deploy the sample driver provided when you install the WebLogic RFID Edge Server software. The sample includes java files that demonstrate how to write custom reader drivers. The sample provided is for the ThingMagic Mercury 4.

## Deploying the Sample Driver

The custom module and any other classes that it uses must be compiled into a jar file. You can deploy the module in to the Edge Server by adding this jar file to the custom modules directory in the Edge Server installation and restarting the server.

To build and deploy the sample reader, use the following steps:

1. Run the build script (C:\bea\rfid_edge30\samples\Drivers\build.bat). This compiles the java files and creates the `custom-drivers.jar file`.
2. Copy the JAR file to the `RFID_EDGE_HOME/plugin/driver` directory.
3. Restart the Edge Server

   The Edge Server loads the driver and makes the device available on the Administration Server device list. The Reader Type

   You may create subdirectories in the driver directories. When the Edge Server starts, it searches recursively through the directories and loads the JAR files that it finds.

# Index

## B