ALIEN TECHNOLOGY®

JAVA DEVELOPER'S GUIDE

September 2008

All Readers

Legal Notices

Alien Technology®

Java Developer's Guide

All Alien RFID Readers

September 2008

Table of Contents

CHAPTER 1
Introduction

The *Java Developer's Guide* provides basic instructions for programmatically controlling a reader using the Java programming language and the class libraries supplied by Alien Technology® as part of the Developers Kit.

Audience

For the purposes of this book, we assume the readers of the *Java Developer's Guide*:

- are competent PC users
- have minimal previous knowledge of radio-frequency identification technology
- are experienced in Java software development

Type Conventions

- Regular text appears in a plain, sans-serif font.
- External files and documents appear in *italic text*.
- Class names appear in a `fixed-width serif font`.
- Things you type in, and sample code appear:

```
indented, in a fixed-width serif font.
```

- Longer blocks of sample code appear:

```
within a shaded, outlined block
in a smaller, fixed-width serif font
```

Overview

The Alien RFID Reader can be programmatically controlled using a number of systems and languages. This document focuses on controlling the reader using the Java library supplied with the developer's kit. This library takes the form of a single JAR file, called "*AlienRFID.jar*", located in the Jars directory on the developer kit CD.

The library contains five discrete functional groups (Java packages) for controlling various aspects of the reader:

- **reader** - classes for communicating with a reader
- **tags** - classes dealing with RFID tags and EPC data
- **discovery** - classes for discovering the locations of readers connected by RS-232 or Ethernet
- **notify** - classes for listening to "push" notifications and streamed data from readers over the LAN

- **util** – classes for performing bit operations, converting between hex/ASCII/binary strings, XML parsers, serial port management, and for determining the API version.

JavaDocs

The automatically generated JavaDoc class documentation for the *AlienRFID.jar* library is also included on the developer kit CD. This documentation set is found in the "doc" directory.

Installation

To use the Alien RFID library from within Java applications and Java applets, the *AlienRFID.jar* file must be added to the Classpath of the application or project. Each development system handles adding Jar files in a different manner and as such the development system documentation should be consulted for guidance on this topic.

System Compatibility

The class files that reside inside the *AlienRFID.jar* file are all Java 1.3 compliant. The library contains no user interface classes, allowing it to run within simple web applets, or full Swing-based Java applications.

API Version

The version number and release date for each build of the AlienRFID.jar file is contained within fields of the static class, com.alien.enterpriseGateway.util.API.

```
package com.alien.enterpriseRFID.util;

public class API {
    public static String version = "2.2.6";
    public static String date = "19 Jun 2008";
    public static String author = "David Krull";
}
```

A Note on Serial Communication

All Alien readers can be operated either via serial communications or over the network. Networking classes are built into the basic Java language and by combining the *AlienRFID.jar* file with a standard Java implementation; all the tools are present to use a reader over the network. However Java classes for serial communications are not part of the standard Java distribution. These are available as a no-cost addition from Sun Microsystems at the following website: http://java.sun.com/products/javacomm/

These serial communications classes (the javax.comm package) must be installed before an Alien reader can be used over serial ports. Complete instructions for installing the javax.comm package on a variety of platforms are included in the download. Installation simply involves placing a handful of files inside your JDK and/or JRE installation.

If the serial classes are not installed, you can still communicate with readers using TCP/IP. In this circumstance, a warning message is issued to stdout when a reader class is instantiated:

```
No Serial Ports Available
The Java Class Libraries for Serial Communication are not
present on this machine.
```

CHAPTER 2
Reader Classes

Introduction

The reader classes are the primary classes for communicating with a reader either over the network or a serial port. Typically the reader object will be obtained from a `DiscoveryItem` object, as discussed in a subsequent chapter. However, if the location (either serial port or network address) of the reader is known, a reader object can be instantiated directly without the need of any discovery classes.

Once a valid reader object is created, it offers you a number of simple commands that fully implement the command set described in the *Reader Interface Guide*.

The parent reader class, `AbstractReader`, provides the base functionality of a generic reader class, handling serial and network communications, and low-level methods for sending commands to a reader and receiving back responses. As its name suggests it is an abstract class and therefore cannot be instantiated, so one of its subclasses is used instead:

- **`AlienClass1Reader`** is the class representing the all of Alien's Class I passive readers, and exposes all of the reader commands of the passive readers through it's API. The vast majority of Alien readers is handled by this single class.
- **`AlienClassOEMReader`** extends `AlienClass1Reader` to communicate with Alien RFID OEM Modules, using a binary protocol. Only serial communication is possible with OEM Modules.
- **`AlienClassBPTReader`** extends `AlienClass1Reader` and includes all of the additional functionality of the Battery Powered Tag reader. We discuss `AlienClassBPTReader` in a later chapter.

Creating a Reader Object from a DiscoveryItem

If discovery classes are used (see subsequent chapters), any readers that are found on the network or serial ports are represented by `DiscoveryItem` objects. To derive a reader object from a `DiscoveryItem`, use the following method:

```
AlienClass1Reader reader = discoveryItem.getReader();
```

If the discovered reader is actually a Battery Powered Tag Reader or OEM Module, the reader object that is returned will actually be of the appropriate class, and can be cast as such:

```
AlienClass1Reader class1Reader = discoveryItem.getReader();

if (class1Reader instanceof AlienClassBPTReader) {
    AlienClassBPTReader classBPTReader = (AlienClassBPTReader)class1Reader;
} else if (class1Reader instanceof AlienClassOEMReader) {
    AlienClassOEMReader classOEMReader = (AlienClassOEMReader)class1Reader;
}
```

Example Code: Recasting a Reader Object

Directly Instantiating a Reader

If it is known that a reader exists on a specified serial port or network address, a new reader object can be created directly without having to use the discovery classes. One way to do this is to instantiate the reader object using the no-arguments constructor, and then use the setConnection() method to specify the name of the serial port it is connected to or its network address.

Example of instantiating a reader on a serial port:

```
AlienClass1Reader reader = new AlienClass1Reader();
reader.setConnection("COM1");
```

Example of instantiating a reader at a network address:

```
AlienClass1Reader reader = new AlienClass1Reader();
reader.setConnection("1.2.3.4", 23);
reader.setUsername("alien");
reader.setPassword("password");
```

Note that the IP address of the reader (as a String) and the port number it uses for network commands (as an integer, typically port 23 – the Telnet port) must both be given. Also, before connecting to a network-based reader, you must specify the username and password using the setUsername() and setPassword() methods. By default all readers use "alien" as the username and "password" as the password.

The AlienClass1Reader class automatically authencates you when you open a socket connection, but you must specify these values first. Failure to set the correct username and password will result in an exception being thrown when trying to connect to a reader on the LAN.

There are other methods used to specify the serial port or network address:

```
AlienClass1Reader.setSerialConnection("COM1");
AlienClass1Reader.setNetworkConnection("1.2.3.4", 23);
AlienClass1Reader.setNetworkConnection("1.2.3.4:23");
```

You can also specify this information in the constructor:

```
AlienClass1Reader reader = new AlienClass1Reader("COM1");
AlienClass1Reader reader = new AlienClass1Reader("1.2.3.4:23");
```

Opening and Closing a Reader Connection

Once you instantiate a reader object and configure its connection settings, you may then send commands to it and ask it for data. To open a reader connection, use the following method:

```
reader.open();
```

If the connection fails, an AlienReaderConnectionException is thrown.

A test can be made on a reader object to see if a connection is already open:

```
reader.isOpen();
```

This method returns `true` if the connection is open, `false` if not. Finally, use the following method to close a reader connection:

```
reader.close();
```

Communicating with a Reader

All commands to and from the reader are ASCII text based messages that take the form of command-response pairs (the OEM Module uses a binary communication protocol, which is hidden from you by the `AlienClassOEMReader` class). The most basic way to communicate with a reader is to use a method called doReaderCommand(), which sends an ASCII command and returns the ASCII response.

```
String readerName = reader.doReaderCommand("get ReaderName");
```

However, this method requires knowledge of the reader command set and requires you to parse and process all reader responses. For instance, in the above example, the readerName variable would be set to "ReaderName = Alien RFID Reader" (the reader's response), which probably isn't what you wanted (just the reader's name).

To make life simpler for the developer, the reader classes provide many additional methods that directly correspond to the reader command set. For example, there is a method called getReaderName(), which returns just the reader's name (not the entire response). Similarly, there is a method called getPersistTime(), that is effectively the same as calling doReaderCommand("get PersistTime") and then parsing the string reply and casting it into an integer value.

AlienClass1Reader class

The `AlienClass1Reader` class provides access to the base command set of all Alien Class 1 RFID readers. Rather than manage individual classes for each reader model, this common class handles all of them. Some methods in `AlienClass1Reader` may not apply to older readers, depending on the firmware revision and model. Attempting to call these methods will likely result in an exception being thrown (see end of this chapter for reader exceptions).

Each of the reader's attributes is exposed by getter and setter methods, as well as other reader commands like notifyNow(), autoModeReset(), programTag(), etc. Additionally, some reader commands can be issued in a number of ways, as provided by the API. For example, nine methods are available to help you set tag masks.

See the JavaDocs for a complete listing of `AlienClass1Reader` methods and their descriptions. There are over 380 public methods in this class alone, so we won't describe each one of them here. The remaining chapters deal with specific API topics, and the appendices outline each of the sample applications provided by the API.

The rest of this chapter describes how to use the AlienClassBTPReader and AlienClassOEMReader classes, which allow you to work with the older ALR-2850 Battery-Powered Tag (BPT) reader and ALR-9930 OEM module.

AlienClassBPTReader class

The Alien Battery Powered Tag Readers support extra commands especially designed to take advantage of the enhanced functionality of Alien's Battery Tags. These extra commands fall into three categories:

Memory

The Battery Tags can optionally support read-write on-board memory, typically in the range of 4K to 16K bytes. The memory commands described in this document allow this tag memory to be read and written in discrete blocks via RF communication.

Sensors
The Battery Tags can optionally support the use of on-board sensors such as temperature or vibration sensors. The sensor commands can be used to interrogate and control the use of these tag devices.

Logging
If a tag is equipped with one or more sensors and on-board memory, it can be instructed to autonomously log data to tag memory even in the absence of an RF field. The logging commands are the interface to this functionality.

Battery Powered Tag Public Methods

As mentioned previously, the methods available to the `AlienClassBPTReader` class directly mirror the command line options for the BPT Reader as discussed in the *Reader Interface Guide*. Following is a brief list of these methods. Please refer to the JavaDocs for full details of these command.

public Tag getTagID(String tagID)
Returns the ID of a unique tag specified by the mask commands.

public String getTagInfo(String tagID)
Returns information about a single tag.

public int getSensorValue(String tagID)
Returns the sensor value of the tag's onboard sensor.

public boolean isLogging(String tagID)
Returns the status of Logging Mode for the specified tag.

public void setLogging(String tagID, boolean isLogging)
Enables or disables logging for a specified tag.

public byte[] getLoggingInterval(String tagID)
Returns the periodicity at which the tag logs sensor data to tag memory.

public void setLoggingInterval(String tagID, int hours, int mins, int secs)
Sets the periodicity at which the tag logs sensor data to tag memory.

public short[] getMemory(String tagID, int lengthIndex, int startIndex)
Returns a chunk of tag memory from the specified tag.

public boolean setMemory(String tagID, int startIndex, byte byteArray[])
Stores a series of bytes into tag memory.

public void clearMemory(String tagID)
Completely erases the memory of a specified tag.

public int getMemoryPacketSize()
Returns the number of bytes to use in each memory packet to and from the tag.

public void setMemoryPacketSize(int memoryPacketSize)
Specifies the number of bytes to use in each memory packet to and from the tag.

AlienClassOEMReader class

Although `AlienClassOEMReader` inherits from `AlienClass1Reader`, its actual implementation is quite different. This is because while all other Alien RFID Readers communicate with an easy-to-read ASCII-based command/response protocol, the OEM Modules use a binary protocol. The difference is clear comparing even a simple command like "look for tags and tell me what you see":

ASCII-Based Protocol Example	
Command	get Taglist
Response	0102 0304 0506 0708 8000 8004 13DB 34DE
Binary Protocol Example	
Command	10 01 21 00 40 00 01 03 03 00 00 01 04 EE 10 02
Response	21 49 40 01 D1 3B 21 49 40 02 00 00 08 01 02 03 04 05 06 07 08 4C D8 21 49 40 02 00 00 08 80 00 80 04 13 DB 34 DE 4C D8 21 49 40 03 00 02 D2 F4 D3 95 CC F4

`AlienClassOEMReader` overrides the send and receive methods of `AlienClass1Reader` taking care of packetizing the command and depacketizing the reader's response. It also overrides the reader command methods that apply to the OEM Module, replacing the simple ASCII command strings with their counterparts in the binary protocol.

The command set of the OEM Module is a subset of other Alien readers, and some operations like AutoMode and NotifyMode are missing. To compensate for this, some of this high-level functionality it taken over by `AlienClassOEMReader`. For instance, the class implements a very basic version of AutoMode – turning it on causes the class to repeatedly ask the reader to read for tags, storing the detected tags in a taglist internal to the class (rather than being stored on the reader itself. Calling getTagList() while AutoMode is on returns this internal taglist, rather than issuing an acquire command to the reader.

`AlienClassOEMReader` also supports a basic command-line interface, such as that provided by the Gateway demonstration software. The Command Line Interface tool allows you talk to an OEM Module using the same (though more restricted) ASCII protocol:

```
Alien>help
*********************************************
*
* Help
*
*********************************************
GENERAL:
    Help (H)
    Info (I)
    ! (Repeat Last Command)
    Get ReaderName
    Get ReaderType
    Get ReaderVersion
    Get MfgInfo
    Get|Set AntennaSequence (i, j, k...)
    Get|Set Antenna
    Get ExternalInput
    Set ExternalOutput
TAGLIST:
    Get|Set AcquireMode
    Get|Set AcqCycles
    Get|Set AcqCount
    Get|Set AcqEnterWakeCount
    Get|Set AcqExitWakeCount
    Get|Set AcqSleepCount
    Get|Set PersistTime (secs)
    Get TagList {n}
    Clear TagList
    Wake
    Sleep
    Get|Set Mask (All | bitLen, bitPtr, XX XX)
AUTONOMOUS MODE:
    Get|Set Automode (On or Off)
    AutoModeReset
PROGRAMMING:
    Program Tag = XX XX XX XX XX XX XX XX
    Verify Tag
    Erase Tag
    Kill Tag = XX XX XX XX XX XX XX XX YY
    Lock Tag = YY
MISC:
    Get Timer
    Set ReaderCommand = XX XX XX...
    RC = XX XX XX...

(XX = TagID byte)
(YY = LockCode byte)

Alien>get taglist
Tag=0102 0304 0506 0708  Disc=Fri Jun 18 12:58:18 PDT 2004  Last=Fri Jun 18
12:58:18 PDT 2004  Ant=0  Count=3

Alien>set AntennaSequence = 0,1
AntennaSequence (i, j, k...) = 0, 1
```

Example: Using Comand Line Interface Application With OEM Module

This is not to imply that you can connect a host to an OEM Module and communicate with terminal software using the ASCII command protocol – but the capability exists for the `AlienClassOEMReader` class to react to text-based commands. It does this by overriding `AbstractReader`'s sendString() and receiveString() methods, catching strings that look like known commands and handling them with other methods that are private to the class.

AlienDLEObject class

The binary protocol used for communicating with the OEM Module is not nearly as simple to use as the ASCII protocol. Commands, arguments, and return values are all specified as sequences of bytes, and are packetized for transmission to and from the OEM

Module. This packetization involves adding ReaderNumber and SessionID bytes to the command/response payload, and then DLE (Data Link Escape) and SOM (Start of Message) tokens at the beginning of a message, and corresponding EOM (End of Message) and DLE tokens at the end of the message (hence the name, `AlienDLEObject`). Furthermore, CRC bytes must be calculated and added onto the packet for verification of transmission.

These tasks are made easier with the `AlienDLEObject` class, which provides constants for all of the commands, subcommands, and response codes that the OEM Module understands. The prepareGenericCommand() series of methods allow you to simply specify the command and optional arguments and will packetize the command for you.

```
AlienClassOEMReader reader = new AlienClassOEMReader("COM1");

AlienDLEObject command = new AlienDLEObject();

// Turn all external digital outputs off
command.prepareGenericCommand(readerCommand.CMD_SET_IO_PORT_VALUE, 0);

reader.issueReaderCommand(command);
```

Example Code: Creating and Issuing a DLE Command

But where is the reader's response? When `AlienClassOEMReader`'s issueReaderCommand() method is done sending the command, it waits for the reader's response and fills in a number of fields in the `AlienDLEObject` object with the data.

public byte[] replyBuffer;
This is where the raw bytes of the reader's reply are stored.

public int replyCommType;
The CommType is the reader's response code, which indicates success (0x00), some other non-error condition (<0x80), or an error condition (>0x80).

public String replyCommTypeMessage;
Returns a human-readable string describing the CommType.

public int replyValueInt;
If the response data is a single byte (the ReaderNumber, for instance), it will be be cast to an int and stored here for quick access.

public byte[] replyValueHexArray;
All of the response data bytes are stored here for quick access.

public int[] replyValueIntArray;
All of the response data bytes are converted to integers and stored here for quick access.

Alien Reader Class Exceptions

The classes in the com.alien.enterpriseRFID.reader package handle error circumstances by throwing exceptions back to the calling object. There are six reader exceptions, all extending from a single `AlienReaderException` class:

- `AlienReaderCommandErrorException`
- `AlienReaderConnectionException`
- `AlienReaderInvalidArgumentException`

- `AlienReaderNoTagException`
- `AlienReaderNotValidException`
- `AlienReaderTimeoutException`

AlienReaderCommandErrorException
This exception may be thrown if the reader responds to a command with an error.

AlienReaderConnectionException
This exception may be thrown if there is a problem connecting to a reader. For example, if a serial connection is attempted but the serial port is already in use by another application, or a network connection is attempted but the reader can't be found on the network.

AlienReaderInvalidArgumentException
This exception may be thrown by the AlienClassOEMReader class if a command is sent to the reader with arguments out-of-range or an improper number of arguments.

AlienReaderNoTagException
This exception may be thrown by AlienClassBPTReader if a command is sent to a specific Battery Tag, but the tag cannot be found.

AlienReaderNotValidException
This exception may be thrown by any of the reader classes if a connection is attempted but the device on the other end isn't an Alien RFID Reader.

AlienReaderTimeoutException
This exception may be thrown by any of the reader classes if any reader transaction takes longer than the timeout duration. The default timeout is 10 seconds, but this can be changed with the setTimeOutMilliseconds () method.

CHAPTER 3
Tag Classes

Introduction

Tags play a very important part in the RFID reader and tag system. For this reason there is a single class devoted to storing and manipulating tag information: the Tag class. Additional classes and an interface are helpful for managing raw tag data and tag lists within your own applications.

Reading Tags

The reader classes allow tags to be read by the reader and return the results either as a single Tag object or an array of Tag objects:

```
Tag[] tagList = reader.getTagList();
```

You may also get tag data from a reader by receiving a notification message from it (see next chapter). In either case, each returned tag is represented by a `Tag` object, which encapsulates not just the tag's ID but also data telling where, when, and how many times the tag was reader

Tag class

The `Tag` class has the following members, each of which is accessible through getters and setters in the API:

- **Tag ID** – a string representing the tag's EPC code
- **Discover Time** – the time the tag was first seen by the reader
- **Last Seen Time** – the time the tag was last seen by the reader
- **Count** – the number of times the reader has read the tag since it was first seen
- **Antenna** – the (transmit) antenna number where the tag was last seen
- **Protocol** – the air protocol used to acquire the tag's ID
- **TransmitAntenna & ReceiveAntenna** – Multi-static antenna systems, such as the ALR-9800 can indicate which antenna was transmitting, and which antenna was receiving when the tag was acquired.
- **G2Data** – The reader can optionally fetch additional tag data besides the tag's EPC.
- **RSSI** – The reader can optionally report the tag's Return Signal Strength Indication.
- **Speed & SmoothSpeed** – The reader can optionally measure the tag's speed. Repeated measurements can generate a smoothed speed value.
- **SmoothPosition** – Repeated speed measurements integrated over time gives you a measurement of the tag's position.

- **Direction** – whether the tag is approaching or receeding from the antenna (requires speed data)

Tag Public Methods

Some of the more useful public methods provided by `Tag` are listed below.

public String getTagID()
Returns the tag's ID.

public int getAntenna()
Returns the antenna that this `Tag` was last seen on.

public long getDiscoverTime()
Returns the time this `Tag` was first seen by the reader.

public long getRenewTime()
Returns the time this `Tag` was last seen be the reader.

public int getRenewCount()
Returns the number of times this `Tag` has been read.

public int getRSSI()
Returns the last RSSI measurement for this tag.

public int getSmoothSpeed()
Returns the averaged/smoothed speed of this tag.

public int getSmoothPosition()
Returns the tag's position, relative to its first read.

public int getDirection()
Returns the apparent direction of the tag, toward or away from the antenna.

TagUtil class

Static methods for parsing and decoding raw tag list data are available in `TagUtil`:

public static Tag[] decodeTagList(String tagList)
Decodes a text-based tag list message into an array of Tags.

public static Tag decodeTag(String tagData)
Decodes a single text-based line of tag list data into a single Tag item.

public static Tag[] decodeXMLTagList(String xmlTagList)
Decodes an XML-based tag list message into an array of Tags.

public static Tag decodeXMLTag(String xmlTagData)
Decodse an individual tag's information from an XML-based tag message.

public static Tag[] decodeCustomTagList(String tagList, String customFormatString)
Decodes a custom-formatted taglist message into an array of Tags, using the supplied TagListCustomFormat definition.

public static Tag decodeCustomTag(String tagLine, String customFormatString)
Decodes a single custom-formmated line of tag list data into a single Tag item, using the supplied TagListCustomFormat definition.

public static void setCustomFormatString(String customFormatString)
Preloads the TagUtil class with the reader's current TaglistCustomFormat string. It uses this to later decode custom-formatted taglist data quickly.

public static Tag[] decodeCustomTagList(String tagLine)
Decodes a custom-formatted taglist message into an array of Tags, using the last supplied TagListCustomFormat definition.

TagTable class

`TagTable` maintains a `HashTable` of tags, with methods for adding and removing tags, as well as hooks for notifying your application about changes to the list:

public boolean addTag(Tag tag)
Adds a `Tag` to this `TagTable`.

public boolean removeTag(Tag tag)
Removes a `Tag` from this `TagTable`.

public boolean removeOldTags()
Removes `Tags` from this `TagTable` whose TimeToLive has reached zero.

public int getPersistTime()
Returns this `TagTable`'s persistence time.

public void setPersistTime(int persistTime)
Specifies the persistence time for tags in this `TagTable`.

public Tag[] getTagList()
Returns the list of `Tags` in this `TagTable`, as an array of `Tag` objects.

public TagTableListener getTagTableListener()
Returns the object that has been registered with this `TagTable` to receive events when the list changes.

public void setTagTableListener(TagTableListener tagTableListener)
Registers a `TagTableListener` with this `TagTable` to be notified when the list changes.

TagTableListener interface

This interface used to communicate `TagTable` changes to other objects. It requires three methods be implemented:

```
public void tagAdded(Tag tag);
public void tagRenewed(Tag tag);
public void tagRemoved(Tag tag);
```

An object that implements this interface can be registered with a `TagTable` and these methods would be called whenever tags are added to, updated, or removed from, the `TagTable`.

CHAPTER 4
Discovery Classes

Introduction

In order to use and control a reader, its network address or the serial port number on the host where it is connected must first be known. The *AlienRFID.jar* library provides the classes, `SerialDiscoveryListenerService` and `NetworkDiscoveryListenerService`, needed to automatically search for and discover readers using both of these connection modes.

In both cases, the service is created, an object is registered as the recipient of discovery events, and the service is started. Once started, the service runs on its own thread until it stops automatically (for serial discovery) or is told to stop (for network discovery).

DiscoveryListener interface

In order for an object to receive discovery events from one of the discovery service classes, it must implement the `DiscoveryListener` interface. This interface is define as follows:

```
public interface DiscoveryListener {
    public void readerAdded  (DiscoveryItem discoveryItem);
    public void readerRenewed(DiscoveryItem discoveryItem);
    public void readerRemoved(DiscoveryItem discoveryItem);
}
```

Definition of the DiscoveryListener interface

When a discovery service runs and discovers a for the first time reader, it calls the readerAdded() method of the registered listener. Knowledge of this reader is maintained by the service, and if the same reader is discovered again, the service calls the listener's readerRenewed() method. If the discovery service loses track of a reader, the listener's readerRemoved() method is called.

Each method is handed a single argument, a `DiscoveryItem`. To retrieve an array of known readers, call the discovery service's getDiscoveryItems() method.

DiscoveryItem class

This class contains key information points that allow any software system to identify and contact the discovered reader - information such as the ReaderName, ReaderType and address. The `DiscoveryItem` provides this information through a series of getters and setters, but one value method exists to translate this information into a usable reader class:

```
public AlienClass1Reader getReader() throws Exception
```

Calling this method on a `DiscoveryItem` returns an `AlienClass1Reader` object, which the object used to directly interface with a reader, and is described in the following chapter.

DiscoveryItem Public Methods

Some of the more useful public methods provided by `DiscoveryItem` are listed below.

public String getReaderName()
Returns this name of the discovered reader.

public String getReaderType()
Returns the type of the discovered reader.

public String getReaderAddress()
Returns the address of the discovered reader.

public String getReaderMACAddress()
Returns the MAC address of the discovered reader, if provided by that reader (null otherwise).

public String getConnection()
Returns the connection method of the reader, "serial" or "network".

public int getCommandPort()
Returns the port number that the discovered reader uses to accept host commands over the network.

public int getLeaseTime()
Returns the amount of time until the discovered reader is due to send another heartbeat message.

public long getFirstHeartbeat()
Returns the time that this `DiscoveryItem` first registered a heartbeat signal from its reader.

public long getLastHeartbeat()
Returns the time that this `DiscoveryItem` last registered a heartbeat signal from its reader.

public String getReaderVersion()
Returns ReaderVersion string of the discovered reader.

public AlienClass1Reader getReader()
As discussed above, this method creates an `AlienClass1Reader` object from the `DiscoveryItem`.

SerialDiscoveryListenerService class

Discovery of a reader attached to the serial port of a host computer is simply a case of checking every serial port for the presence of an Alien reader. This is achieved using the `SerialDiscoveryListenerService` class, as in the following code example:

```
import com.alien.enterpriseRFID.discovery.*;

public class SerialDiscoveryTest implements DiscoveryListener {

public SerialDiscoveryTest() {
    SerialDiscoveryListenerService service = new SerialDiscoveryListenerService();
    service.setDiscoveryListener(this);
    service.startService();
    ...(application continues)...
}

public void readerAdded(DiscoveryItem discoveryItem) {
    System.out.println("Added:\n" + discoveryItem.toString());
}

public void readerRenewed(DiscoveryItem discoveryItem) {
    System.out.println("Renew:\n" + discoveryItem.toString());
}

public void readerRemoved(DiscoveryItem discoveryItem) {
    System.out.println("Removed:\n" + discoveryItem.toString());
}
```

Example Code: Serial Discovery (from SerialDiscoveryTest.java)

When the `SerialDiscoveryListenerService` object is instantiated and started, it automatically acquires a list of all the serial ports on the host computer and then proceeds to interrogate each port, looking for a reader. If a reader is found, or on subsequent scans a reader is lost or renewed, the appropriate `DiscoveryListener` method is called.

Additionally, an `ActionListener` can be registered with the `SerialDiscoveryListenerService`. The actionPerformed() method is called when the `SerialDiscoveryListenerService` scans each port, and can be used as a progress monitor. See the example code in the Developer's Kit for more information.

The default implementation of the Java platform does not ship with classes to communicate over serial ports. This functionality must be downloaded and installed from Sun Microsystems at the following website: http://java.sun.com/products/javacomm/.

If the serial library is not installed, instantiating a `SerialDiscoveryListenerService` throws an exception with the message "Serial Discovery Instance Failed - Serial Classes Not Present".

SerialDiscoveryListenerService Public Methods

Some of the more useful public methods provided by `DiscoveryItem` are listed below. The public methods for `NetworkDiscoveryListenerService` are the same as for `SerialDiscoveryListenerService`.

public void setDiscoveryListener(DiscoveryListener discoveryListener)
Registers an object with the discovery service to receive messages when readers are discovered, renewed, or removed.

public void startService()
Starts up the discovery service.

public void stopService()
Stops/pauses the discovery service.

public boolean isRunning()
Returns true if the discovery service is still running.

public DiscoveryItem[] getDiscoveryItems()
Returns an array of `DiscoveryItems` representing all of the readers that the discovery service knows about.

public void setMaxSerialPort(int maxPort)
Set the maximum COM port number to scan to.

public void setSerialPortList(String portList)
Specifies a comma-separated list of COM port numbers to scan.

NetworkDiscoveryListenerService class

Each Alien reader is configured, by default, to broadcast heartbeat messages over its local subnet. These messages are UDP (User Datagram Protocol) packets containing small XML documents which detail the reader's type, name, and contact information. By listening for these heartbeat messages, the network discovery classes can identify and report back details of readers that exist on the network.

The class that performs these listening duties is called `NetworkDiscoveryListenerService`. Once this class is instantiated and started, it will run in its own thread until it is stopped. While running, it listens for reader heartbeats on the listener port (which is specified in the constructor or defaults to 3988), calling either the readerAdded() or readerRenewed() methods of a registered `DiscoveryListener` when it detects a reader. Part of the heartbeat sent out by the reader indicates the time until the next heartbeat is expected. If this time expires before the next heartbeat is received, then the service assumes the reader has gone offline and will call the readerRemoved() method.

Following is a code example demonstrating how to perform network discovery:

```
import com.alien.enterpriseRFID.discovery.*;

public class NetworkDiscoveryTest implements DiscoveryListener {

public NetworkDiscoveryTest() throws Exception {
    NetworkDiscoveryListenerService service = new
NetworkDiscoveryListenerService();
    service.setDiscoveryListener(this);
    service.startService();
    ...(application continues)...
}


public void readerAdded(DiscoveryItem discoveryItem) {
    System.out.println("Added:\n" + discoveryItem.toString());
}

public void readerRenewed(DiscoveryItem discoveryItem) {
    System.out.println("Renew:\n" + discoveryItem.toString());
}

public void readerRemoved(DiscoveryItem discoveryItem) {
    System.out.println("Removed:\n" + discoveryItem.toString());
}
```

Example Code: Network Discovery (from NetworkDiscoveryTest.java)

NetworkDiscoveryListenerService Public Methods

The public methods for `NetworkDiscoveryListenerService` are the same as for `SerialDiscoveryListenerService` (see above).

Discovery Service Exceptions

The classes in the com.alien.enterpriseRFID.discovery package handle error circumstances by throwing exceptions back to the calling object. There are three discovery exceptions, all extending from a single `AlienDiscoveryException` class:

- `AlienDiscoverySerialException`
- `AlienDiscoverySocketException`
- `AlienDiscoveryUnknownReaderException`

AlienDiscoverySerialException

This exception may be thrown by a SerialDiscoveryListenerService if the serial classes are not present.

AlienDiscoverySocketException

This exception may be thrown by a NetworkDiscoveryListenerService if it is unable to bind to the specified heartbeat listener port. This is likely because another application has bound to the same port (perhaps another discovery service).

AlienDiscoveryUnknownReaderException

This exception may be thrown when trying to create a reader object from a `DiscoveryItem` through its getReader() method. If the ReaderType contained in the DiscoveryItem is now a recognized type, then the correct type of reader object cannot be created, and the method fails.

CHAPTER 5
Notify Classes

Introduction

The notify classes work in conjunction with a reader running in autonomous mode. In autonomous mode the reader is configured to read tags over and over again without the need for human interaction. The reader can be configured to send messages to listening services on the network when specific events occur, such as a timer expiring, tags added/removed from the taglist, successful/unsuccessful programing, etc.

The notify classes implement such a listening services, constantly waiting and listening for notification messages from readers, and converting these messages into Java objects which are then available to your application.

The same notify classes that listen for periodic notification messages from the reader will also handle streamed data from the reader – the TagStream and IOStream modes push tag- and I/O- events to the listener as they happen on the reader, providing low-latency data now achievable through the traditional notification mechanism.

MessageListenerService class

The key class in the notify package is `MessageListenerService`. This is a service that listens at a specified port for incoming reader notification messages. Following is the basic code showing how to use the `MessageListenerService`:

```
import com.alien.enterpriseRFID.notification.*;

public class MessageListenerTest implements MessageListener  {

public MessageListenerTest() throws Exception {
    MessageListenerService service = new MessageListenerService(3988);
    service.setMessageListener(this);
    service.startService();
    ...(application continues)...
}


public void messageReceived(Message message) {
    System.out.println("Message Received: " + message.toString());
}
```

Example Code: Using the MessageListenerService

The `MessageListenerService` is set up to listen to a specified port number, and the reader (or multiple readers) can then be set up to notify the service on that port. This is done as follows:

1. Instruct the reader to send notification messages to the host running the `MessageListenerService`. For example, if the service is running on a machine named "listener.alien.com", the following reader command would be issued:

   ```
   reader.setNotifyAddress("listener.alien.com:3988");
   ```

2. Instruct the reader about the conditions which should trigger a notification messages. For example, to tell the reader to send out a tag list every 30 seconds, the following command could be issued:

```
reader.setNotifyTime(30);
```

3. Set the notification message format to XML. In order for `MessageListenerService` to be able to decode the notification message for you, it must be in "XML" or "Text" format:

```
reader.setNotifyFormat(reader.XML_FORMAT);
```

4. Finally tell the reader to start reading tags in autonomous mode. For example to tell the reader to read as fast as it can until told otherwise, the following two commands could be issued:

```
reader.autoModeReset();
reader.setAutoMode(reader.ON);
```

At this point the reader switches into autonomous mode and, as instructed, sends out a message every 30 seconds to the `MessageListenerService`, containing its internal tag list and additional information about the notification.

The listener service as set up above constantly listens for these messages on its own thread until told to stop. When a message is received, it is parsed and converted into a `Message` object, which is passed to the messageReceived() method of the registered `MessageListener`.

MessageListenerService Public Methods

public int getListenerPort()
Returns the port number to listening on for incoming notification messages.

public void setListenerPort(int listenerPort)
Specifies the port number to listening on for incoming notification messages.

public MessageListener getMessageListener()
Returns the `MessageListener` registered to receive notification events.

public void setMessageListener(MessageListener messageListener)
Registers a `MessageListener` to receive notification events.

public void startService()
Starts the `MessageListenerService`.

public void stopService()
Stops the `MessageListenerService`.

public boolean isRunning()
Returns true if the `MessageListenerService` is running.

public void setIsCustomTagList()
Flags whether to use the custom taglist decoder or the standard "Text" format decoder when decoding tag data.

MessageListener interface

In order for an object to receive notification events from a `MessageListenerService`, it must implement the `MessageListener` interface. This interface is define as follows:

```
public interface MessageListener{
    public void messageReceived(Message message);
}
```

Definition of the MessageListener interface

Only one method to implement, and it simply receives a `Message` object from the `MessageListenerService`.

Message class

A `Message` object encapsulates a collection of metadata about the notification message itself, and an array of `Tag` objects extracted from the taglist portion of the notification message. It contains the following members, all of which are available through getter and setter accessor methods:

- **ReaderName** – the name of the reader
- **ReaderType** – the type of reader
- **IPAddress** – the IP address of the reader
- **MACAddress** – the MAC address of the reader, if provided
- **CommandPort** – the port number on which to send commands to the reader
- **Time** – the date and time the message was sent out
- **Reason** – why the message sent out by the reader
- **StartTriggerLines** – indicates which external inputs triggered the reader to start
- **StopTriggerLines** – indicates which external inputs triggered the reader to stop
- **TagList** - an array of `Tag` objects extracted from the notification

Message Public Methods

public String getReaderName()
Returns the name of the reader that sent the notification message.

public String getReaderType()
Returns the type of the reader that sent the notification message..

public String getReaderIPAddress()
Returns the IP Address of the reader that sent the notification message.

public int getReaderCommandPort()
Returns the Command port number of the reader that sent the notification message.

public String getReaderMACAddress()
Returns the MAC address of the reader that sent the notification message, if provided by that reader (null otherwise).

public Date getDate()
Returns the date and time of the Message.

public String getReason()
Gets the reason why the reader send the Message.

public int getStartTriggerLines()
Returns the external input lines that triggered this autonomous cycle to start.

public int getStopTriggerLines()
Returns the external input lines that triggered this autonomous cycle to start.

public int getTagCount()
Returns the number of tags in the message's TagList.

public Tag[] getTagList()
Returns the TagList of the notification message, as an array of `Tags`.

public Tag getTag(int index)
Returns the `Tag` that holds position index in the message's TagList.

public String getRawData()
Returns the raw content of the notification message, before decoding.

public int getIOCount()
Returns the number of I/O evenbts in the message's IOList.

public ExternalIO[] getIOList()
Returns the IOList from the notification message, as an array of `ExternalIOs`.

public ExternalIO getIO(int index)
Returns the `ExternalIO` that holds position index in the message's IOList.

ErrorMessage class

An `ErrorMessage` object is used by the `MessageListenerService` to communicate any problems it had while trying to receive or decode a notification message from a reader. ErrorMessage extends Message, so it can by handed off to a MessageListener's MessageReceived() method just the same.

The ErrorMessage will return useful information about the error though the following methods:

- **getReaderIPAddress()** – which reader was sending the message
- **getReason()** – the reason for the error
- **getRawData()** – the raw data that was received by the reader.

To handle these conditions, use an "instanceof" construct in your MessageReceived() as follows:

```
public void MessageReceived(Message m) {
    if (message instanceof ErrorMessage) {
        // message is bad
        System.out.println("Notify error from " + m.getReaderIPAddress());
        System.out.println("Problem is: " + m.getReason());
        System.out.println("Data read is: " + m.getXML());
    } else {
        // message is good
        ...
    }
}
```

Message Class Exceptions

The classes in the com.alien.enterpriseRFID.notify package handle error circumstances by throwing exceptions back to the calling object. There are two notify exceptions, all extending from a single `AlienMessageException`class:

- `AlienMessageConnectionException`
- `AlienMessageFormatException`

AlienMessageConnectionException

This exception may be thrown if the reader encounters a communication error during the receipt of a message from a reader.

AlienMessageFormatException

This exception may be thrown if the notification message is not in XML format, or there is some problem parsing the XML.

Appendix A
Annotated Examples

Introduction

The following examples are taken from the example source code distributed in the API. The examples are already throughly commented, but this appendix will provide more information on how the code is used.

The following copyright statement applies to each of the source code examples, and is stated here once for brevity:

```
/**
 * Copyright 2008 Alien Technology Corporation. All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1)   Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2)   Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * 3)   Neither the name of Alien Technology Corporation nor the names of any
 * contributors may be used to endorse or promote products derived from this
 * software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL ALIEN TECHNOLOGY CORPORATION OR ITS CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * For further information, contact :
 * Alien Technology
 * 18220 Butterfield Blvd.
 * Morgan Hill, CA 95037
 */
```

AlienClass1ReaderTest.java

This example opens a connection to a reader on COM1, fetches it's TagList, and prints the results.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.reader.*;
4   import com.alien.enterpriseRFID.tags.*;
5
6   /**
7    * Connects to a Reader on COM port #1 and asks it to read tags.
8    *
9    * @version 1.2 Feb 2008
10   * @author David Krull
11   */
12  public class AlienClass1ReaderTest {
13
14  /**
15   * Constructor
16   */
17  public AlienClass1ReaderTest() throws AlienReaderException {
18
19      AlienClass1Reader reader = new AlienClass1Reader();
20      reader.setConnection("COM1");
21
22      // To connect to a networked reader instead, use the following:
23      /*
24          reader.setConnection("10.1.60.107", 23);
25          reader.setUsername("alien");
26          reader.setPassword("password");
27      */
28
29      // Open a connection to the reader
30      reader.open();
31
32      // Ask the reader to read tags and print them
33      Tag[] tagList = reader.getTagList();
34      if (tagList == null) {
35          System.out.println("No Tags Found");
36      } else {
37          System.out.println("Tag(s) found:");
38          for (int i=0; i<tagList.length; i++) {
39              Tag tag = tagList[i];
40              System.out.println("ID:" + tag.getTagID() +
41                  ", Discovered:" + tag.getDiscoverTime() +
42                  ", Last Seen:" + tag.getRenewTime() +
43                  ", Antenna:" + tag.getAntenna() +
44                  ", Reads:" + tag.getRenewCount()
45              );
46          }
47      }
48
49      // Close the connection
50      reader.close();
51  }
52
53  /**
54   * Main
55   */
56  public static final void main(String args[]){
57      try {
58          new AlienClass1ReaderTest();
59      } catch(AlienReaderException e) {
60          System.out.println("Error: " + e.toString());
61      }
62  }
```

```
63
64  } // End of class AlienClass1ReaderTest
```

Lines 1-4 – Define the package for this example, and import the required "reader" and "tags" packages from the library.

Lines 12, 17 – Define the class and constructor for this example.

Lines 19-27 – Instantiate a new AlienClass1Reader object for a reader found on serial port COM1. Lines 24-26 show how to do the same thing with a reader on the network. You must set the username and password properties in the reader object before attempting to open() it over a TCP connection, since the class needs to authenticate with the reader at connection time.

Line 30 – Opens a connection with the reader. This claims ownership of the serial port, or opens a TCP socket to the reader's CommandPort, depending on the connection type.

Line 33 – Asks the reader to read tags and report back the TagList, parses the resulting String, and hands you back an array of Tag objects, which we are storing in the tagList array.

Lines 34-35 – Checks to see if tagList is null – the result you get when there are no tags, and prints an appropriate message.

Lines 36-47 – Prints a "Tags found" message, then loops through each of the Tags in tagList, printing out the EPC, discovery and last-seen times, antenna, and read count. There are also convenient Tag.toString() and Tag.toLongString() methods.

Line 50 – Closes the connection to the reader. This releases the serial port, or closes down the TCP socket. Since only once connection (of each method) can be made to a reader at a time, it is a good idea to close the connection when you are done, to prevent unwanted blocking of the reader's command channel. The reader may also disconnect your TCP socket if it has been idle longer than the NetworkTimeout, or if another TCP connection is made to the reader from the same IP address.

Lines 56-61 – The main function, which creates the AlienClass1ReaderTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
Tag(s) found:
ID:0000 0000 0000 0000 0000 0006, Discovered:1220395010000, Last Seen:
1220395010000, Antenna:1, Reads:1
ID:0000 0000 0000 0000 0000 000D, Discovered:1220395010000, Last Seen:
1220395010000, Antenna:0, Reads:1
ID:0000 0000 0000 0000 0000 0002, Discovered:1220395010000, Last Seen:
1220395010000, Antenna:0, Reads:1
ID:0000 0000 0000 0000 0000 0007, Discovered:1220395010000, Last Seen:
1220395010000, Antenna:0, Reads:1
ID:0000 0000 0000 0000 0000 000E, Discovered:1220395010000, Last Seen:
1220395010000, Antenna:0, Reads:1
```

AlienClass1Communicator.java

This example opens a connection to a reader on COM1, and begins an interactive telnet-style session, taking input from the keyboard, sending it to the reader, and printing the results. Type "q" to quit.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.reader.*;
4   import java.io.*;
5
6   /**
7    * Connects to a Reader on COM port #1 and begins an interactive session.
8    * Enter "q" to quit the session.
9    *
10   * @version 1.1 Feb 2004
11   * @author David Krull
12   */
13  public class AlienClass1Communicator {
14
15  /**
16   * Constructor.
17   */
18  public AlienClass1Communicator() throws Exception {
19      AlienClass1Reader reader = new AlienClass1Reader("COM1");
20      reader.open(); // Open the reader connection
21
22      // Use stdin for user input
23      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
24
25      do {
26          System.out.print("\nAlien>"); // Show prompt
27          String line = in.readLine(); // Grab user input
28          if (line.equals("q")) break; // Quit when "q" is pressed
29          System.out.println(reader.doReaderCommand(line));
30      } while (true); // Repeat indefinitely
31
32      System.out.println("\nGoodbye.");
33      reader.close(); // Close the reader connection
34  }
35
36
37  /**
38   * Main
39   */
40  public static final void main(String args[]){
41      try {
42          new AlienClass1Communicator();
43      } catch(Exception e) {
44          System.out.println("Error: " + e.toString());
45      }
46  }
47
48  } // End of class AlienClass1Communicator
```

Lines 1-4 – Define the package for this example, and import the required "reader" and "java.io" packages from the library.

Lines 13, 18 – Define the class and constructor for this example.

Lines 19-20 – Instantiate a new AlienClass1Reader object for a reader found on serial port COM1, and opens a connection to the reader.

Line 23 – Creates a BufferedReader, to assist in reading input from stdin one line at a time.

Lines 25,30 – Sets up an endless do-while loop – the main part of the program.

Line 26 – Prints out the "Alien>" prompt for the user.

Lines 27-28 – Reads a line from stdin, breaking out of the loop if the user enters "q".

Line 29 – Sends what the user typed to the reader, using the generic doReaderCommand() method. The results are printed to the screen, exactly as they were received from the reader.

Line 33 – After exiting the do-while loop, we close the connection to the reader and allow the program to exit.

Lines 40-46 – The main function, which creates the AlienClass1Communicator object, and catches and prints all exceptions generated therein.

Sample Output:

```
Alien>get ReaderName
ReaderName = David's 9800

Alien>i network
**************************************************************
NETWORK COMMANDS
**************************************************************
  MACAddress = 00:80:66:10:11:6A
  DHCP = ON
  DHCPTimeout = 90
  IPAddress = 10.10.82.72
  Hostname = r72
  UpgradeAddress = http://10.10.82.10/dailybuild/
  NetworkUpgrade = ON
  Gateway = 10.10.82.1
  Netmask = 255.255.255.0
  DNS = 10.1.1.2
  NetworkTimeout = 65535
  CommandPort = 23
  HeartbeatAddress = 255.255.255.255
  HeartbeatPort = 3988
  HeartbeatTime = 3
  HeartbeatCount = -1
  WWWPort = 80
  AcceptConnections = ANY

Alien>t
Tag:0000 0000 0000 0000 0000 000D, Disc:2008/09/02 15:45:01, Last:2008/09/02
15:45:01, Count:1, Ant:0, Proto:2
Tag:0000 0000 0000 0000 0000 000E, Disc:2008/09/02 15:45:01, Last:2008/09/02
15:45:01, Count:1, Ant:0, Proto:2
Tag:0000 0000 0000 0000 0000 0007, Disc:2008/09/02 15:45:01, Last:2008/09/02
15:45:01, Count:1, Ant:0, Proto:2
Tag:0000 0000 0000 0000 0000 0002, Disc:2008/09/02 15:45:01, Last:2008/09/02
15:45:01, Count:1, Ant:0, Proto:2
Tag:0000 0000 0000 0000 0000 0006, Disc:2008/09/02 15:45:01, Last:2008/09/02
15:45:01, Count:1, Ant:1, Proto:2

Alien>
```

SerialDiscoveryTest.java

This example demonstrates how to scan all of the serial ports on the system, looking for Alien RFID readers.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.discovery.*;
4   import java.awt.event.*;
5
6   /**
7    * Starts a SerialDiscoveryService to scan the local serial ports and look for
8    * Alien Nanoscanner Readers.
9    *
10   * @version 1.1 November 2003
11   * @author David Krull
12   */
13  public class SerialDiscoveryTest implements DiscoveryListener, ActionListener
14  {
15
16  /**
17   * Constructor.
18   */
19  public SerialDiscoveryTest() throws Exception {
20      SerialDiscoveryListenerService service;
21      service = new SerialDiscoveryListenerService();
22      service.setDiscoveryListener(this);
23      service.setActionListener(this);
24      service.startService();
25      while (service.isRunning()) {
26          Thread.sleep(100);
27      }
28  }
29
30
31  /**
32   * A reader has been discovered on a serial port. This method implements the
33   * DiscoveryListener interface.
34   */
35  public void readerAdded(DiscoveryItem discoveryItem) {
36      System.out.println("Reader Added:\n" + discoveryItem.toString());
37  }
38
39  /**
40   * A known reader has been seen again. This method implements the
41   * DiscoveryListener interface, but doesn't really apply to Serial Discovery.
42   */
43  public void readerRenewed(DiscoveryItem discoveryItem) {
44      System.out.println("Reader Renewed:\n" + discoveryItem.toString());
45  }
46
47
48  /**
49   * A reader has been removed from the network and is no longer available or
50   * valid. This method implements the DiscoveryListener interface, but doesn't
51   * apply to serial discovery.
52   */
53  public void readerRemoved(DiscoveryItem discoveryItem) {
54      System.out.println("Reader Removed:\n" + discoveryItem.toString());
55  }
56
57
58  /**
59   * ActionEvents are sent by the SerialDiscoveryListenerService to this method
60   * while it scans the serial ports. This is mainly for the purposes of
61   * debugging and displaying on-screen progress to the user. The ActionEvents
62   * are sent when the discovery service starts scanning each port, and again
```

```
63   * when it is all done scanning.
64   */
65  public void actionPerformed(ActionEvent event) {
66      if (event.getID() == SerialDiscoveryListenerService.SCANNING PORT) {
67          System.out.println("Scanning Serial Port:" + event.getActionCommand());
68      }
69      if (event.getID() == SerialDiscoveryListenerService.SCANNING_END) {
70          System.out.println("Scanning Finished");
71          System.out.println("Total Readers found = "
72                  + service.getDiscoveryItems().length);
73      }
74  }
75
76
77  /**
78   * Main
79   */
80  public static final void main(String args[]) {
81      try {
82          new SerialDiscoveryTest();
83      } catch (Exception e) {
84          System.out.println("Error:" + e.toString());
85      }
86  }
87
88  } // End of class SerialDiscoveryTest
```

Lines 1-4 – Define the package for this example, and import the required "discovery" and "java.awt.event" packages from the library. java.awt.event is only needed because we wish to see the activity of the SerialDiscoveryListenerService.

Lines 13, 19 – Define the class and constructor for this example. This application implements the DiscoveryListener interface (for receiving reader discovery events) and also ActionListener (for receiving activity events from the SerialDiscoveryListenerService).

Lines 20-21 – Instantiate a new SerialDiscoveryListenerService object.

Line 22 – Tells the service to notify our app (which implements the DiscoveryListener interface) when readers are discovered.

Line 23 – Tells the service to notify our app of its activity (optional).

Line 24 – Starts the discovery service. It runs on its own now, and will use callbacks to let us know what is happening.

Lines 25-27 – Sets up an endless do-while loop – we will sit and wait for callbacks from the service (need to ctrl-C to exit the program).

Lines 35-55 – Implement the DiscoveryListener interface – three methods, readerAdded() for when a reader is first discovered, and readerRenewed() and readerRemoved(), which aren't applicable for serial discovery. The SerialDiscoveryListenerService will call readerAdded() each time it finds a reader on a COM port. It passes along a DiscoveryItem containing all of the information required to connect to the new reader, which we print out.

Lines 65-74 – Implement the ActionListener interface. This is optional, and simply provides feedback, in the form of ActionEvents, to indicate when the SerialDiscoveryListenerService examines each serial port, and when it stops. event.getID() tells you which event it is, and you can compare it against some constants in the SerialDiscoveryListenerService class. We print out a message each time a port is scanned, and a final message indicating how many readers were discovered. service.getDiscoveryItems() returns an array of DiscoveryItems for each of the readers discovered on serial ports. You can alternatively

start the service, wait a period of time, and then query the service for the list, instead of implementing the DiscoveryListener interface.

Lines 80-86 – The main function, which creates the SerialDiscoveryTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
Scanning Serial Port: COM1

Reader Added:
Reader Name      = David's 9800
Reader Type      = Alien RFID Tag Reader, Model: ALR-9800 (Four Antenna / Multi-
Protocol / 902-928 MHz)
Reader Address   = COM1
Reader MACAddress = 00:80:66:10:11:6A
Connection       = Serial
Command Port     = 0
Lease Time       = 10000
Discovery Method = Automatic

Scanning Serial Port: COM2

Scanning Serial Port: COM3

Reader Added:
Reader Name      = David's 9780
Reader Type      = Alien RFID Tag Reader, Model: ALR-9780 (Four Antenna / EPC
Class 1 Gen 2/ 915 MHz)
Reader Address   = COM3
Reader MACAddress = 00:90:c2:c2:71:d3
Connection       = Serial
Command Port     = 0
Lease Time       = 10000
Discovery Method = Automatic

Scanning Finished
Total Readers found = 2
```

NetworkDiscoveryTest.java

This example demonstrates how to run a service that discovers Alien readers on the network.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.discovery.*;
4
5   /**
6    * Starts a NetworkDiscoveryService to listen for Alien Reader.
7    * heartbeats that are broadcast over the local subnet. The discovery service
8    * notifies this application when a reader is discovered, seen again, or lost.
9    *
10   * @version 1.3 Aug 2008
11   * @author David Krull
12   */
13
14  public class NetworkDiscoveryTest implements DiscoveryListener {
15
16  /**
17   * Constructor.
18   */
19  public NetworkDiscoveryTest() throws Exception {
20      NetworkDiscoveryListenerService service;
21      service = new NetworkDiscoveryListenerService();
22      service.setDiscoveryListener(this);
23      service.startService();
24
25      // Spin while readers are discovered.
26      while (service.isRunning()) {
27          Thread.sleep(100);
28      }
29  }
30
31
32  /**
33   * A new reader has been discovered to the network.
34   * This method implements the DiscoveryListener interface.
35   */
36  public void readerAdded(DiscoveryItem discoveryItem){
37      System.out.println("Reader Added:\n" + discoveryItem.toString());
38  }
39
40  /**
41   * A known reader has been seen again.
42   * This method implements the DiscoveryListener interface.
43   */
44  public void readerRenewed(DiscoveryItem discoveryItem) {
45      System.out.println("Reader Renewed:\n" + discoveryItem.toString());
46  }
47
48
49  /**
50   * A reader has been removed from the network and is no longer available.
51   * This method implements the DiscoveryListener interface.
52   */
53  public void readerRemoved(DiscoveryItem discoveryItem) {
54      System.out.println("Reader Removed:\n" + discoveryItem.toString());
55  }
56
57
58  /**
59   * Main.
60   */
61  public static final void main(String args[]) {
62      try {
```

```
63          new NetworkDiscoveryTest();
64      } catch (Exception e) {
65          System.out.println("Error:" + e.toString());
66      }
67  }
68
69  } // end of class NetworkDiscoveryTest
```

Lines 1-3 – Define the package for this example, and import the required "discovery" package from the library.

Lines 14, 19 – Define the class and constructor for this example. This application implements the DiscoveryListener interface for receiving reader discovery events from the NetworkDiscoveryListenerService.

Lines 20-21 – Instantiate a new NetworkDiscoveryListenerService object.

Line 22 – Tells the service to notify our app (which implements the DiscoveryListener interface) when readers are discovered.

Line 23 – Starts the discovery service. It runs on its own now, and will use callbacks to let us know what is happening.

Lines 26-28 – Sets up an endless do-while loop – we will sit and wait for callbacks from the service (need to ctrl-C to exit the program).

Lines 36-55 – Implement the DiscoveryListener interface – three methods, readerAdded() for when a reader is first discovered, readerRenewed() for when subsequent heartbeats are received from a reader that is already known about, and readerRemoved() for when the service stops receiving regular heartbeats from a known reader. We print out a simple message for each event, including the DiscoveryItem that is passed to each method.

Lines 61-67 – The main function, which creates the NetworkDiscoveryTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
// Two readers are first discovered…
Reader Added:
Reader Name      = Reader1
Reader Type      = Alien RFID Tag Reader, Model: ALR-9800 (Four Antenna / Multi-
Protocol / 902-928 MHz)
Reader Address   = 10.10.82.72
Reader MACAddress = 00:80:66:10:11:6A
Reader Version    = 08.09.02.00b
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic

Reader Added:
Reader Name      = Reader2
Reader Type      = Alien RFID Tag Reader, Model: ALR-9650 (One Antenna / Gen 2 /
902-928 MHz)
Reader Address   = 10.10.82.233
Reader MACAddress = 0A:0B:0C:0D:0E:22
Reader Version    = 08.06.26.00
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic

// Now we see heartbeats from the same two readers again…
Reader Renewed:
Reader Name      = Reader1
```

```
Reader Type      = Alien RFID Tag Reader, Model: ALR-9800 (Four Antenna / Multi-
Protocol / 902-928 MHz)
Reader Address   = 10.10.82.72
Reader MACAddress = 00:80:66:10:11:6A
Reader Version    = 08.09.02.00b
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic

Reader Renewed:
Reader Name      = Reader2
Reader Type      = Alien RFID Tag Reader, Model: ALR-9650 (One Antenna / Gen 2 /
902-928 MHz)
Reader Address   = 10.10.82.233
Reader MACAddress = 0A:0B:0C:0D:0E:22
Reader Version    = 08.06.26.00
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic

// Now the 1st reader has been disconnected - we still get heartbeats from the 2nd…
Reader Renewed:
Reader Name      = Reader2
Reader Type      = Alien RFID Tag Reader, Model: ALR-9650 (One Antenna / Gen 2 /
902-928 MHz)
Reader Address   = 10.10.82.233
Reader MACAddress = 0A:0B:0C:0D:0E:22
Reader Version    = 08.06.26.00
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic

// Finally, we get the message that the 1st reader is gone…
Reader Removed:
Reader Name      = Reader1
Reader Type      = Alien RFID Tag Reader, Model: ALR-9800 (Four Antenna / Multi-
Protocol / 902-928 MHz)
Reader Address   = 10.10.82.72
Reader MACAddress = 00:80:66:10:11:6A
Reader Version    = 08.09.02.00b
Connection      = Network
Command Port    = 23
Lease Time      = 30
Discovery Method = Automatic
```

MessageListenerTest.java

This example demonstrates how to run a service that accepts notification messages pushed out by Alien readers on the network. It sets up the reader at COM1 in AutoMode, with NotifyMode configured to send regular tag-read messages back to our application, where the messages are printed out.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.reader.*;
4   import com.alien.enterpriseRFID.tags.*;
5   import com.alien.enterpriseRFID.notify.*;
6
7   import java.net.InetAddress;
8
9   /**
10   * Starts up a message listener service, then opens a connection to a reader
11   * connected to COM1 and tells sets up autonomous mode. The reader sends a
12   * message to this application every second, whether it sees a tag or not.
13   * <p>
14   * The notifications are passed to the messageReceived method, where the tag
15   * list is then displayed.
16   * <p>
17   * This application will run for 10 seconds, and then it will reconnect to the
18   * reader and turn off AutoMode and NotifyMode. If you don't exit this
19   * application nicely, say with a ctrl-C or similar method, the reader is
20   * still reading and notifying, even though the application has exited.
21   * <p>
22   * The solution to this is to log onto the reader and turn AutoMode off.
23   *
24   * @version 1.3 July 2008
25   * @author David Krull
26   */
27  public class MessageListenerTest implements MessageListener  {
28
29  /**
30   * Constructor.
31   */
32  public MessageListenerTest() throws Exception {
33      // Set up the message listener service
34      MessageListenerService service = new MessageListenerService(4000);
35      service.setMessageListener(this);
36      service.startService();
37      System.out.println("Message Listener has Started");
38
39      // Instantiate a new reader object, and open a connection to it on COM1
40      AlienClass1Reader reader = new AlienClass1Reader("COM1");
41      reader.open();
42      System.out.println("Configuring Reader");
43
44      // Set up Notification. Use this host's IPAddress, and the port number
45      // that the service is listening on.
46      // getHostAddress() may find a wrong (wireless) Ethernet interface, so you
47      // may need to substitute your computers IP address manually.
48      String myIP = InetAddress.getLocalHost().getHostAddress();
49      reader.setNotifyAddress(myIP, service.getListenerPort());
50
51      // Make sure service can decode it.
52      reader.setNotifyFormat(AlienClass1Reader.XML_FORMAT);
53      // Notify whether there's a tag or not
54      reader.setNotifyTrigger("TrueFalse");
55      reader.setNotifyMode(AlienClass1Reader.ON);
56
57      // Set up AutoMode
58      reader.autoModeReset();
59      reader.setAutoStopTimer(1000); // Read for 1 second
```

```
60      reader.setAutoMode(AlienClass1Reader.ON);
61
62      // Close the connection and spin while messages arrive
63      reader.close();
64      long runTime = 10000; // milliseconds
65      long startTime = System.currentTimeMillis();
66      do {
67          Thread.sleep(1000);
68      } while(service.isRunning()
69              && (System.currentTimeMillis()-startTime) < runTime);
70
71      // Reconnect to the reader and turn off AutoMode and TagStreamMode.
72      System.out.println("\nResetting Reader");
73      reader.open();
74      reader.autoModeReset();
75      reader.setNotifyMode(AlienClass1Reader.OFF);
76      reader.close();
77  }
78
79
80  /**
81   * A single Message has been received from a Reader.
82   */
83  public void messageReceived(Message message) {
84      System.out.println("\nMessage Received:");
85      if (message.getTagCount() == 0) {
86          System.out.println("(No Tags)");
87      } else {
88          for (int i=0; i<message.getTagCount(); i++) {
89              Tag tag = message.getTag(i);
90              System.out.println(tag.toLongString());
91          }
92      }
93  }
94
95
96  /**
97   * Main.
98   */
99  public static final void main(String args[]){
100     try {
101         new MessageListenerTest();
102     } catch (Exception e) {
103         System.out.println("Error:" + e.toString());
104     }
105 }
106
107 } // end of class MessageListenerTest
```

Lines 1-7 – Define the package for this example, and import the required "discovery", "tags", and "notify" packages from the library. We also need to deduce out own IP address, so we import java.net.InetAddress too.

Lines 27, 32 – Define the class and constructor for this example. This application implements the MessageListener interface for receiving reader notification events from the MessageListenerService.

Line 34 – Instantiate a new MessageListenerService object. We tell it which port (4000) to listen on. The reader needs to know out IP address and this port number in order to deliver notification messages.

Line 35 – Tell the service to notify our app (which implements the MessageListener interface) when reader notifications are received.

Line 36 – Start the listener service. It runs on its own now, and will use callbacks to let us know when data is received from a reader.

Lines 40-41 – Open a connection to the reader at COM1, so that we can configure it to send us messages.

Line 48 – Get out computer's IP address. If you have more than one Ethernet interface (wireless, or VPN software, for instance) this method may get the IP address from the wrong interface. You may need to hardcode your computer's IP address here.

Line 49 – Set up the reader's NotifyAddress property to be of the format: <myIPAddress>:<listenerPort>.

Lines 52-55 – Set the format of the reader's notification messages to XML, tell the reader to notify us no matter what happens, and turn on NotifyMode.

Lines 58-63 – Reset the reader's AutoMode settings to the defaults, turn AutoMode on, and disconnect from the reader.

Lines 64-69 – Set up a timer to wait 10 seconds, or until the MessageListenerService stops. The listener service will run on its own, waiting for readers to connect to it's port. It will then read the data from the reader, decode any tag or I/O data there, and hand you a Message object via the messageReceived() method.

Lines 72-76 – After the expiration of the timer, we connect back to the reader to turn AutoMode and NotifyMode off. Otherwise, the reader will continue to run even after our app has exited.

Lines 82-93 – Implement the MessageListener interface, consisting of a single method, messageReceived(). The MessageListenerService passes us a Message object, which contains information about the reader as well as a TagList (and IOList, in case the notification contained I/O events as well). We print out a simple message the method is called, including the tag data.

Lines 99-105 – The main function, which creates the MessageListenerTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
Message Listener has Started
Configuring Reader

Message Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:01:08 PDT 2008  Last=Wed Sep
03 10:01:09 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:01:08 PDT 2008  Last=Wed Sep
03 10:01:09 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

Message Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:01:09 PDT 2008  Last=Wed Sep
03 10:01:10 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:01:09 PDT 2008  Last=Wed Sep
03 10:01:10 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

// …10 of these messages were received, once per second…

Message Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:01:16 PDT 2008  Last=Wed Sep
03 10:01:17 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:01:16 PDT 2008  Last=Wed Sep
03 10:01:17 PDT 2008  Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=  Disc=Wed Sep 03 10:01:16 PDT 2008  Last=Wed Sep 03 10:01:17 PDT 2008
Count=6  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

Resetting Reader
```

TagStreamTest.java

This example demonstrates how to use a MessageListenerService to receive and deliver to your application streamed data from a reader. Readers can stream data to you everytime a tag is read (TagStream) or everytime an external I/O changes (IOStream). This is just like the MessageListenerTest.java example, only we configure TagStreamMode on the reader instead of NotifyMode. The MessageListenerService works the same.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import com.alien.enterpriseRFID.reader.*;
4   import com.alien.enterpriseRFID.tags.*;
5   import com.alien.enterpriseRFID.notify.*;
6
7   import java.net.InetAddress;
8
9   /**
10   * Starts up a message listener service, then opens a connection to a reader
11   * connected to COM1 and configures it to go into autonomous mode and stream
12   * tag reads back to this application.
13   * <p>
14   * The TagStream events are delivered to the messageReceived method, where the
15   * tag list is then displayed.
16   * <p>
17   * Only enterprise class readers (ALR-x800/9900/9650) support TagStreaming,
18   * and they must have a firmware revision of at least 07.01.31.
19   *
20   * One thing to note: This application will run for 10 seconds, and then
21   * it will reconnect to the reader and turn off AutoMode and TagStreamMode.
22   * If you don't exit this application nicely, say with a ctrl-C or similar
23   * method, the reader is still reading and streaming tags, even though the
24   * application has exited.
25   * <p>
26   * The solution to this is to log onto the reader and turn AutoMode off.
27   *
28   * @version 1.0 July 2008
29   * @author David Krull
30   */
31  public class TagStreamTest implements MessageListener  {
32
33  /**
34   * Constructor.
35   */
36  public TagStreamTest() throws Exception {
37      // Set up the message listener service.
38      // It handles streamed data as well as notifications.
39      MessageListenerService service = new MessageListenerService(4000);
40      service.setMessageListener(this);
41      service.startService();
42      System.out.println("Message Listener has Started");
43
44      // Instantiate a new reader object, and open a connection to it on COM1
45      AlienClass1Reader reader = new AlienClass1Reader("COM1");
46      reader.open();
47      System.out.println("Configuring Reader");
48
49      // Set up TagStream. Use this host's IPAddress, and the port number that
50      // the service is listening on.
51      // getHostAddress() may find a wrong (wireless) Ethernet interface,
52      // so you may need to substitute your computers IP address manually.
53      String myIP = InetAddress.getLocalHost().getHostAddress();
54      reader.setTagStreamAddress(myIP, service.getListenerPort());
55      // Make sure service can decode it.
56      reader.setTagStreamFormat(AlienClass1Reader.TEXT_FORMAT);
57      reader.setTagStreamMode(AlienClass1Reader.ON);
```

```
58
59      // Set up AutoMode - use standard settings.
60      reader.autoModeReset();
61      reader.setAutoMode(AlienClass1Reader.ON);
62
63      // Close the connection and spin while messages arrive
64      reader.close();
65      long runTime = 10000; // milliseconds
66      long startTime = System.currentTimeMillis();
67      do {
68          Thread.sleep(1000);
69      } while(service.isRunning()
70              && (System.currentTimeMillis()-startTime) < runTime);
71
72      // Reconnect to the reader and turn off AutoMode and TagStreamMode.
73      System.out.println("\nResetting Reader");
74      reader.open();
75      reader.autoModeReset();
76      reader.setTagStreamMode(AlienClass1Reader.OFF);
77      reader.close();
78  }
79
80
81  /**
82   * A single Message has been received from a Reader.
83   */
84  public void messageReceived(Message message){
85      System.out.println("\nStream Data Received:");
86      if (message.getTagCount() == 0) {
87          System.out.println("(No Tags)");
88      } else {
89          for (int i=0; i<message.getTagCount(); i++) {
90              Tag tag = message.getTag(i);
91              System.out.println(tag.toLongString());
92          }
93      }
94  }
95
96
97  /**
98   * Main.
99   */
100 public static final void main(String args[]){
101     try {
102         new TagStreamTest();
103     } catch (Exception e) {
104         System.out.println("Error:" + e.toString());
105     }
106 }
107
108 } // End of class TagStreamTest
```

Lines 1-7 – Define the package for this example, and import the required "discovery", "tags", and "notify" packages from the library. We also need to deduce out own IP address, so we import java.net.InetAddress too.

Lines 31, 36 – Define the class and constructor for this example. This application implements the MessageListener interface for receiving reader notification events from the MessageListenerService.

Line 39 – Instantiate a new MessageListenerService object. We tell it which port (4000) to listen on. The reader needs to know out IP address and this port number in order to deliver notification messages.

Line 40 – Tell the service to notify our app (which implements the MessageListener interface) when reader notifications are received.

Line 41 – Start the listener service. It runs on its own now, and will use callbacks to let us know when data is received from a reader.

Lines 45-46 – Open a connection to the reader at COM1, so that we can configure it to send us messages.

Line 53 – Get out computer's IP address. If you have more than one Ethernet interface (wireless, or VPN software, for instance) this method may get the IP address from the wrong interface. You may need to hardcode your computer's IP address here.

Line 54 – Set up the reader's TagStreamAddress property to be of the format: <myAddress>:<listenerPort>.

Lines 56-57 – Set the format of the reader's TagStream output to Text, and turn on TagStreamMode.

Lines 60-64 – Reset the reader's AutoMode settings to the defaults, turn AutoMode on, and disconnect from the reader.

Lines 65-70 – Set up a timer to wait 10 seconds, or until the MessageListenerService stops. The listener service will run on its own, waiting for readers to connect to it's port. It will then read the data streamed from the reader, decode any tag or I/O data there, and hand you a Message object via the messageReceived() method.

Lines 73-77 – After the expiration of the timer, we connect back to the reader to turn AutoMode and TagStreamMode off. Otherwise, the reader will continue to run even after our app has exited.

Lines 84-94 – Implement the MessageListener interface, consisting of a single method, messageReceived(). The MessageListenerService passes us a Message object, which contains information about the reader as well as a TagList (and IOList, in case the notification contained I/O events as well). We print out a simple message the method is called, including the tag data.

Lines 100-106 – The main function, which creates the TagStreamTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
Message Listener has Started
Configuring Reader

Stream Data Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:27:46 PDT 2008  Last=Wed Sep
03 10:27:46 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:27:46 PDT 2008  Last=Wed Sep
03 10:27:46 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

Stream Data Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:27:46 PDT 2008  Last=Wed Sep
03 10:27:46 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:27:46 PDT 2008  Last=Wed Sep
03 10:27:46 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

// …I received about 50 of these messages in the 10-second period…

Stream Data Received:
Tag=E200 3411 B802 0111 0604 7639  Disc=Wed Sep 03 10:27:56 PDT 2008  Last=Wed Sep
03 10:27:56 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0
Tag=ABCD 3412 DF00 0982 3000 5079  Disc=Wed Sep 03 10:27:56 PDT 2008  Last=Wed Sep
03 10:27:56 PDT 2008  Count=1  Ant=0  Proto=2  v=0.0  RSSI=0.0  Dir=0

Resetting Reader
```

IOStreamTest.java

This example demonstrates how to use a MessageListenerService to receive and deliver to your application streamed data from a reader. Readers can stream data to you everytime a tag is read (TagStream) or everytime an external I/O changes (IOStream). This is just like the TagStreamTest.java example, only we configure IOStreamMode on the reader instead of TagStreamMode. The AutoMode setup is a bit mode complicated, since we are using AutoMode to generate many I/O events by setting the AutoWaitOutput, AutoWorkOutput, etc. to different values. The MessageListenerService works the same.

```
1   package com.alien.enterpriseRFID.examples;
2
3   import java.net.InetAddress;
4
5   import com.alien.enterpriseRFID.externalio.ExternalIO;
6   import com.alien.enterpriseRFID.notify.Message;
7   import com.alien.enterpriseRFID.notify.MessageListener;
8   import com.alien.enterpriseRFID.notify.MessageListenerService;
9   import com.alien.enterpriseRFID.reader.AlienClass1Reader;
10
11  /**
12   * Starts up a message listener service, then opens a connection to a reader
13   * connected to COM1 and configures it to go into autonomous mode with various
14   * ExternalOutput settings for each AutoMode state. This generates many I/O
15   * events, which are streamed back to this application.
16   * <p>
17   * The IOStream events are delivered to the messageReceived method, where they
18   * are displayed.
19   * <p>
20   * Only enterprise class readers (ALR-x800/9900/9650) support IOStreaming, and
21   * they must have a firmware revision of at least 07.01.31.
22   *
23   * This application will run for 10 seconds, and then it will reconnect to the
24   * reader and turn off AutoMode and IOStreamMode. If you don't exit this
25   * application nicely, say with a ctrl-C or similar method, the reader is
26   * still reading and streaming tags, even though the application has exited.
27   * <p>
28   * The solution to this is to log onto the reader and turn AutoMode off.
29   *
30   * @version 1.0 July 2008
31   * @author David Krull
32   */
33  public class IOStreamTest implements MessageListener  {
34
35  /**
36   * Constructor.
37   */
38  public IOStreamTest() throws Exception {
39      // Set up the message listener service.
40      // It handles streamed data as well as notifications.
41      MessageListenerService service = new MessageListenerService(4000);
42      service.setMessageListener(this);
43      service.startService();
44      System.out.println("Message Listener has Started");
45
46      // Instantiate a new reader object, and open a connection to it on COM1
47      AlienClass1Reader reader = new AlienClass1Reader("COM1");
48      reader.open();
49      System.out.println("Configuring Reader");
50
51      // Set up IOStream. Use this host's IPAddress, and the port number that
52      // the service is listening on.
53      // getHostAddress() may find a wrong (wireless) Ethernet interface, so you
54      // may need to substitute your computers IP address manually.
```

```
55      String myIP = InetAddress.getLocalHost().getHostAddress();
56      reader.setIOStreamAddress(myIP, service.getListenerPort());
57      // Make sure service can decode it.
58      reader.setIOStreamFormat(AlienClass1Reader.TEXT_FORMAT);
59      reader.setIOStreamMode(AlienClass1Reader.ON);
60
61      // Set up AutoMode - make it blink various outputs.
62      reader.autoModeReset();
63      reader.setAutoWaitOutput(1);  // output #1
64      reader.setAutoWorkOutput(2);  // output #2
65      reader.setAutoTrueOutput(3);  // outputs #1,2
66      reader.setAutoFalseOutput(0); // no outputs
67      reader.setAutoMode(AlienClass1Reader.ON);
68
69      // Close the connection and spin while messages arrive
70      reader.close();
71      long runTime = 10000; // milliseconds
72      long startTime = System.currentTimeMillis();
73      do {
74          Thread.sleep(1000);
75      } while(service.isRunning()
76              && (System.currentTimeMillis()-startTime) < runTime);
77
78      // Reconnect to the reader and turn off AutoMode and TagStreamMode.
79      System.out.println("\nResetting Reader");
80      reader.open();
81      reader.autoModeReset();
82      reader.setIOStreamMode(AlienClass1Reader.OFF);
83      reader.close();
84  }
85
86
87  /**
88   * A single Message has been received from a Reader.
89   */
90  public void messageReceived(Message message){
91      System.out.println("\nStream Data Received:");
92      if (message.getIOCount() == 0) {
93          System.out.println("(No IOs)");
94      } else {
95          for (int i=0; i<message.getIOCount(); i++) {
96              ExternalIO io = message.getIO(i);
97              System.out.println(io.toLongString());
98          }
99      }
100 }
101
102
103 /**
104  * Main.
105  */
106 public static final void main(String args[]){
107     try {
108         new IOStreamTest();
109     } catch (Exception e) {
110         System.out.println("Error:" + e.toString());
111     }
112 }
113
114 } // End of class IOStreamTest
```

Lines 1-7 – Define the package for this example, and import the required "discovery", "tags", and "notify" packages from the library. We also need to deduce out own IP address, so we import java.net.InetAddress too.

Lines 33, 38 – Define the class and constructor for this example. This application implements the MessageListener interface for receiving reader notification events from the MessageListenerService.

Line 41 – Instantiate a new MessageListenerService object. We tell it which port (4000) to listen on. The reader needs to know out IP address and this port number in order to deliver notification messages.

Line 42 – Tell the service to notify our app (which implements the MessageListener interface) when reader notifications are received.

Line 43 – Start the listener service. It runs on its own now, and will use callbacks to let us know when data is received from a reader.

Lines 47-48 – Open a connection to the reader at COM1, so that we can configure it to send us messages.

Line 55 – Get out computer's IP address. If you have more than one Ethernet interface (wireless, or VPN software, for instance) this method may get the IP address from the wrong interface. You may need to hardcode your computer's IP address here.

Line 56 – Set up the reader's IOStreamAddress property to be of the format: <myAddress>:<listenerPort>.

Lines 58-59 – Set the format of the reader's IOStream output to Text, and turn on IOStreamMode.

Lines 62-67 – Setup the reader's AutoMode parameters and AutoMode on. We set various output values for AutoWaitOutput, AutoWorkOutput AutoTrueOutput and AutoFalseOutput so that as AutoMode runs if generates many I/O events. We aren't concerned with tag data in this example, but the same MessageListenerService can be used for Notifications, TagStream, and IOStream.

Lines 71-76 – Set up a timer to wait 10 seconds, or until the MessageListenerService stops. The listener service will run on its own, waiting for readers to connect to it's port. It will then read the data streamed from the reader, decode any tag or I/O data there, and hand you a Message object via the messageReceived() method.

Lines 79-83 – After the expiration of the timer, we connect back to the reader to turn AutoMode and IOStreamMode off. Otherwise, the reader will continue to run even after our app has exited.

Lines 90-100 – Implement the MessageListener interface, consisting of a single method, messageReceived(). The MessageListenerService passes us a Message object, which contains information about the reader as well as an IOList (and TagList, in case the notification/stream contained Tag events as well). We print out a simple message when the method is called, followed by all of the I/O events.

Lines 106-112 – The main function, which creates the TagStreamTest object, and catches and prints all exceptions generated therein.

Sample Output:

```
Message Listener has Started
Configuring Reader

Stream Data Received:
DO, Value=1, Time=2008/09/03 11:02:34.792, HostTime=2008/09/03 12:02:34.928
DO, Value=2, Time=2008/09/03 11:02:34.792, HostTime=2008/09/03 12:02:34.928

Stream Data Received:
DO, Value=0, Time=2008/09/03 11:02:35.790, HostTime=2008/09/03 12:02:35.913
DO, Value=1, Time=2008/09/03 11:02:35.790, HostTime=2008/09/03 12:02:35.913
DO, Value=2, Time=2008/09/03 11:02:35.791, HostTime=2008/09/03 12:02:35.913

Stream Data Received:
DO, Value=0, Time=2008/09/03 11:02:36.820, HostTime=2008/09/03 12:02:36.975
```

```
DO, Value=1, Time=2008/09/03 11:02:36.820, HostTime=2008/09/03 12:02:36.975
DO, Value=2, Time=2008/09/03 11:02:36.820, HostTime=2008/09/03 12:02:36.975

// …10 of these messages were received, once per second…

Stream Data Received:
DO, Value=0, Time=2008/09/03 11:02:44.411, HostTime=2008/09/03 12:02:44.551
DO, Value=1, Time=2008/09/03 11:02:44.411, HostTime=2008/09/03 12:02:44.551
DO, Value=2, Time=2008/09/03 11:02:44.411, HostTime=2008/09/03 12:02:44.551

Resetting Reader
```