Developing a device adapter and agent for the WebSphere RFID solution, Part 1: Introduction to the WebSphere RFID Device Kit

Skill Level: Intermediate

Karl Freburger (karlf@us.ibm.com)
IT Architect
IBM

Allen Smith (allens@us.ibm.com)
Senior Software Engineer
IBM

13 Jan 2007

This article is the first in a four-part series on building hardware device adapters and agents for the IBM® WebSphere® RFID solution. In this article, you'll learn about the functions provided by the WebSphere RFID Device Infrastructure component and how to construct device adapters using the WebSphere RFID Device Infrastructure Device Development Kit.

WebSphere RFID solution overview

The IBM WebSphere RFID solution provides an infrastructure for processing RFID events. It includes components that interact with RFID devices such as readers and printers, control I/O devices, filter and process RFID events, and send data to and receive data from back-end systems. The WebSphere RFID solution comprises two components: the IBM WebSphere RFID Premises Server Version 1.1.0.1 and the WebSphere RFID Device Infrastructure Version 1.1.1, as shown in Figure 1.

The WebSphere RFID Device Infrastructure (hereafter called Device Infrastructure) provides components that interact with and control RFID hardware devices. Collectively, the Device Infrastructure components are often referred to as an Edge because they are deployed close to the RFID devices on the edge of the network.
These components obtain tag reads from RFID readers, build and submit print jobs to RFID printers, and control I/O devices such as light trees. In addition to interacting with RFID devices, the RFID Device infrastructure components filter and aggregate RFID events such as tag reads and forward these events to the WebSphere Premises Server (hereafter called Premises Server).

The Premises Server receives RFID events from one or more Device Infrastructure Edge devices, then aggregates and further processes the events and makes them available to back-end applications. It provides an integration point for RFID-enabled applications to access and act on RFID events. For example, an application might process a tag read event provided by the Premises Server and, based on the tag ID contained in the event, determine whether the product affixed with the RFID tag was expected. It could then execute an API on the Premises Server to accept the read event, which would cause the Premises Server to instruct the appropriate Edge device to switch the light tree from yellow to green.

The Premises Server is a Java™ 2 Extended Edition (J2EE) based solution that runs on WebSphere Application Server. It includes a Service Management Framework (SMF) component, a bridge, that acts as an intermediary between the J2EE applications and the Device Infrastructure Edges. Communication to and from an Edge is via messages sent through the bridge on the Premises Server (provided by the Premises SMF component) and a bridge on the Edge. The bridge on the Premises Server translates messages between the Edge message format and the Premises Server JMS format. Translated messages received from Edges are placed in an MQ message queue and sent to the Premises Server J2EE applications for processing. JMS messages destined for an Edge are received from Premises Server J2EE applications via an MQ queue, translated by the bridge, and sent to the appropriate Edge bridge for distribution to the Edge components for processing.

Figure 1. WebSphere RFID solution
The WebSphere RFID solution provides two starter kits, or usage scenarios, which can be extended to provide additional capabilities to meet differing customer requirements. The starter kits shipped with the WebSphere RFID solution are:

- The Dock Door Receiving starter kit provides the functions necessary for receiving and processing items with attached RFID tags. These functions include interacting with a RFID reader to receive tag read events, filtering and aggregating the reads, validating the reads and providing visual and audio indicators signaling whether the reads have been accepted or rejected.

- The Print, Verify, and Ship starter kit provides the functions necessary for printing RFID labels, associating cases with containers, validating that outgoing containers contain the correct cases, and generating shipping reports.

For more information on the Websphere RFID solution, refer to the Websphere RFID Information Center, and the Websphere RFID Premises Server product overview (see Resources).

A closer look at RFID Device Infrastructure

RFID Device Infrastructure provides both a runtime and a development environment. The runtime includes a number of device adapters built by IBM that support readers and printers from leading RFID vendors, as shown in Figure 2. These adapters provide the necessary functions for communicating with and executing commands...
on the hardware device. Each adapter has an associated agent that contains logic specific to the WebSphere RFID solution usage scenarios. The runtime includes a number of other agents that perform various functions required by the WebSphere RFID solution usage scenarios. These functions range from filtering and aggregating tag reads to controlling I/O devices such as light trees and motion sensors.

**Figure 2. Device infrastructure runtime**

The Device Infrastructure runtime includes a publication and subscription engine called the MicroBroker. Agents communicate with one another via messages that flow through the MicroBroker. A message is published to a specific topic. An agent subscribes to topics and publishes messages to topics relevant to its functionality. A reader agent for example, would publish a message indicating that a tag read event has occurred to a topic such as `device/reader/L1/signal/tags`. An agent responsible for filtering tag read events would subscribe to that topic, filter out the duplicate tag reads, and publish a different message for each unique tag read to another topic, such as `receiving/portal/L1/signal/tags`.

The MicroBroker has a bridge component that allows messages to be sent from one MicroBroker to another via the MQTT protocol. This bridge is the mechanism used to send messages between an Edge and a Premises Server. As previously mentioned, the Premises Server SMF component includes a MicroBroker bridge. Each bridge, Premises Server, and Edge is configured with the messages that can be sent to and
received by it. The Premises Server bridge is configured with additional settings that instruct it to invoke functions that translate messages received from the Edge format (topic/value) to an XML format and put the messages in a JMS message queue.

The Device Infrastructure components are implemented in Java and run on the IBM Service Management Framework (SMF). SMF is an implementation of the Open Services Gateway Initiative (OSGi) 3.0 specification.

The WebSphere RFID Device Infrastructure is not sold directly by IBM, but is made available to selected partners for embedding on RFID devices such as readers and controllers. When a customer implements the WebSphere RFID solution, they receive the Device Infrastructure through the purchase of a Device Infrastructure-enabled device.

Device infrastructure tools

Device infrastructure includes a development environment called the RFID Tracking Kit. It is installed as a feature in WebSphere Studio Device Developer (hereafter called Device Developer) and includes additional plug-ins and components to facilitate the development of device adapters and agents that can be plugged into the WebSphere RFID solution. Included in the Tracking Kit are:

- An SMF plug-in, which provides SMF tools and a runtime environment that allow you to develop, install, run, test and debug SMF applications, called bundles, in Device Developer.
- An OSGi Application Framework (OAF) plug-in, which provides tools and classes that encapsulate OSGi functions, making it easier to develop OSGi applications.
- A Device Kit plug-in, which provides tools and classes that enable you to build adapters to hardware devices.
- A MicroBroker plug-in, which provides the MicroBroker engine.
- A MicroBroker Application Framework (MBAF) plug-in, which provides classes that encapsulate many of the MicroBroker functions making it easier to develop MicroBroker applications.
- Java projects, including source code, for all the Device Infrastructure agents, which can be loaded into Device Developer and used to extend the behavior of an agent, or used as models for new agents.
- Source code for all the Device Infrastructure adapters, which can be used to extend the behavior of an adapter, or as a model for creating a new one.
• WebSphere Everyplace Custom Environment (hereafter called Everyplace Custom Environment) custom Java configurations.

• An RFID Tracking Kit help plug-in, integrated with Device Developer's help, for each component of the Tracking Kit.

The RFID Tracking Kit provides all the components included in the RFID Device Infrastructure runtime allowing you to run and debug the Device Infrastructure Edge components in Device Developer.

To install the RFID Tracking Kit in Device Developer, refer to the WebSphere RFID Information Center (see Resources). The instructions direct you to install the Tracking Kit feature from a CD-ROM; if you do not have the Tracking Kit on CD-ROM, you can install it from the Device Infrastructure update site (see Resources) using the Device Developer Update Manager.

**Implementing Device Infrastructure components for a device**

To use a new RFID device (such as a tag reader) with the IBM WebSphere RFID solution, you need to implement a new Device Infrastructure agent and device adapter for the device. This series of articles shows you how to do that for the Sirit™ INfinity 510 RFID tag reader.

In general, the process is:

1. Implement the device adapter using the Device Infrastructure Device Kit. You need to implement a Device Kit transport component and device component for your hardware device.

2. Implement the device agent that implements the usage scenario in which your hardware device participates (Dock Door Receiving or Print, Verify, and Ship).

**Device Developer Kit overview**

As previously mentioned, the Device Kit is a component of the Device Infrastructure. It is installed as a plug-in when you install the RFID Tracking Kit in Device Developer. The Device Kit is used to build adapters that communicate with RFID devices, such as readers and printers, as well as non-RFID hardware, like GPS devices or audio components on an automobile entertainment bus. Building an adapter is the first step when integrating a new hardware device into the WebSphere RFID solution.
A Device Kit adapter can be segmented into three components or layers: a connection, a transport and a device as shown in Figure 3.

**Figure 3. Device Kit adapter**

### Connections

A connection, as the name implies, provides a connection to a hardware device. In addition to establishing a communication channel with a hardware device, a connection sends data to (writes bytes) and receives data from (reads bytes) the device. The Device Kit includes several types of connections; the ones of most interest for RFID hardware devices are:

- TcpipConnection for TCP/IP client connections
- SerialConnection for serial (RS232) port connections

When integrating a new hardware device into Device Infrastructure, you can use an existing connection class. You don’t have to write any new code unless you are implementing a new kind of connection.

### Transports
While the connection does the actual sending and receiving of bytes to and from the hardware device, the transport is responsible for converting between bytes and more meaningful objects called messages. When the transport receives data from its connection, it translates the data into one or messages and notifies any listeners (usually device components) that might be interested in messages received by the transport. When sending a message to the hardware device, the transport converts the message into a set of bytes the hardware device can understand. It then invokes one of the connection's send methods to actually send the bytes.

In addition to handling messages, the transport initiates the connection to the hardware device using the appropriate connection class and controls any start-up conversation required by the device. For example, a hardware device might respond with a log-in prompt when a connection is made to it. In this situation, the transport would interpret the bytes received as a log-in prompt and respond with the appropriate set of bytes, typically a user ID and password, to log in into the device. The transport is also responsible for sending heartbeat messages to keep the connection to a hardware device alive.

A transport does not understand the messages it receives and sends. It understands the hardware device protocol enough to parse messages from the input bytes and to ensure that output bytes have the necessary checksums, and so forth. The only messages that a transport should really know about are those that are used in start-up conversations and heartbeats.

When integrating a new hardware device into Device Infrastructure, you'll have to write Java code to implement a transport specific to that device. Implementing the transport is often the most work.

**Devices**

The device component models a hardware device's protocol using objects referred to as commands, measurements and signals. Commands, signals and measurements are collectively referred to as controls. As you might expect, commands are used to model a hardware device's commands. Signals are used to model the messages received from the hardware device. Messages are typically received as the result of executing commands, but may also be asynchronous. Measurements are used to model a property or attribute of the hardware device.

When integrating a new hardware device into Device Infrastructure, you model the device component using Control Markup Language (see CML) and generate the code to implement the controls; you don't write any Java code.

**Device controls: Commands**
A command provides a mechanism for an application to send a request to a hardware device to perform an operation. Some examples are:

- Command to get the software version
- Command to set the operational mode of the reader
- Command to get the tags read by the reader
- Command to set the attenuation of an antenna

Applications can add themselves as listeners of a command. After adding itself as a listener, an application will be notified whenever the command request is sent to the hardware device.

**Device controls: Signals**

A signal is an asynchronous notification that a message has been received from a hardware device. When the transport picks up data from its connection, it converts the received bytes into a message. If the message data maps to an input message that has been modeled by a signal, that signal is triggered. Applications can add themselves as listeners of a signal. After adding itself as a listener, an application will be notified whenever the signal is triggered (the input message is received).

Let’s look at the Sirit INfinity 510 event.tag.report event to illustrate how signals work. The ASCII text representation of this event resembles the following:

```
event.tag.report tag_id=0x3000214160C0040000A5937, antenna=1\r\n\r
```

The above string is sent from the reader every time a tag is singulated and the reader is configured to operate in autonomous or polled mode. The device component defines a signal for this event. The signal includes a message definition that maps the first part of the event string which is `event.tag.report`. Only the first part of the event string is mapped because the `tag_id` (and possibly `antenna`) value will change for each singulated tag. The transport receives (from the connection) the bytes for the `event.tag.report` event sent from the reader, converts them into a message, and delivers them to the device component. The device component checks to determine if the message matches any defined signals. The message received by the transport contains the string `event.tag.report` and the signal control contains a message with the string `event.tag.report`, so there is a match. The signal is triggered and all registered listeners of that signal are notified. If an application has registered an interest in this signal, it is notified and sent the data contained in the `event.tag.report`. 
Device controls: Measurements

Measurements provide applications with a convenient mechanism for getting and setting property values on a hardware device. Measurements are implemented using commands and signals. They combine the functions of these two controls into a single convenient control. Suppose an application wanted to get the software version of a hardware device. The application would have to execute the command that gets the software version and register an interest in the signal that maps the response from the software version command. Alternatively, the application could use a measurement control and call the measurement's read method to get the software version. The underlying handling of the command and signal are done by the measurement. Measurements also maintain a local cached copy of the property value. This feature provides convenient access to hardware device properties whose values are unlikely to change. The software version is a good example of such a property. Applications can add themselves as listeners of a measurement. After adding itself as a listener, an application will be notified each time the measurement's local copy of the property changes.

Application, adapter and hardware device interactions

Figure 4 shows a high-level view of the interactions between the agent (the application), the Device Kit adapter, and the hardware device that occur when the agent uses a command to execute a function on the hardware device. In this example, the following interactions occur:

1. The agent gets the command from the device component.
2. The agent gets the signal mapped to the command response from the device component.
3. The agent adds itself as a listener to the signal.
4. The agent tells the command to execute.
5. The command writes the message to the transport.
6. The transport translates the message into the format expected by the reader and writes the translated message to the connection.
7. The connection sends the bytes to the reader.
8. The connection reads the response bytes from the reader.
9. The transport receives the response from the connection and converts it
to a message.

10. The transport notifies the device component that a message has been received.

11. The message is matched to the message defined for the signal and all listeners of the signal are notified. In this example, the listener is the agent.

Figure 4. Interactions between components

Control Markup Language (CML)

CML is a Device Kit component that is used to model the command set of a hardware device. It is an XML 1.0-compliant mark-up language that defines the controls, messages, and configuration settings required by a device adapter and agent to communicate with and control the hardware device. There is a CML file for the transport component and one for the device component. In the case of transports, a CML file is used to define the messages (both requests and responses) that the transport requires to handle start-up conversations and heartbeats. For device components, a CML file is used to define all the controls (commands, signals and measurements) that need to be available to an application in order to
accomplish a certain task; for example, those commands of a RFID reader that need to be defined in order to tell a reader to start and stop reading and get tag read reports.

The Device Kit tools use the contents of the CML files to generate Java code for the both the transport and device components. For the device component, all the necessary Java code is generated from the CML file. As a result, when building a Device Kit adapter, all the work on the device component is done using CML. For transports, the tools use the CML file to generate the Java definitions of the messages that the transport requires for start-up conversations and heartbeats and the definitions of various transport parameters.

We'll look at some specific examples to get a better understanding of how CML is used to model the command set for a hardware device. Specifically, we'll look at the CML definitions needed to create the controls that model the Sirit INfinity 510 version.sw (software version) command. The information in Figure 5 and Figure 6 comes from the "INfinity 510 Protocol Reference Guide".

Figure 5. INfinity 510 interface protocol

**Interface Protocol**

The INfinity 510 reader’s native protocol is ASCII text based and is used to communicate over the Command/Response and Event Channels.

Every command is sent to the reader over port 50007 and should end with \r\n. The corresponding reader response will end with \r\n\r\n. An example of a command and response is shown in the following:

```
<-setup.region\r
->fcc\r\n\r\n```

Figure 6. INfinity 510 version.sw command
A `<command>` element defines a command. A command definition includes a *message definition*, which specifies the bytes that get sent to the hardware device to execute the command. Listing 1 shows the CML required to model the INfinity 510 version.sw command. Looking at the message definition, you'll notice it contains the string `version.sw`\r\n. The message specifies both the command and the command terminator.

Listing 1. CML to model a command

```cml
<command id="VersionSwGetRequest">
  <message id="VersionSwGetRequestMessage">
    <ascii>version.sw\r\n</ascii>
  </message>
</command>
```

A `<signal>` element defines a signal. A signal definition includes a *template* message definition that specifies the bytes sent by the hardware that should trigger the signal. Listing 2 shows the CML required to model the INInfinity 510's response to the version.sw command. The response from the reader for this command is `ok 0.2.171\r\n`. `ok` is the command response, `0.2.171` is the software version and `\r\n` is the response terminator. Looking at the message definition, you'll notice it contains the command response and response terminator but not the software version. The reason for this is that the software version may vary from reader to reader. We want to define a signal that can be triggered any time a reader responds to the version.sw command, not just when the version is `0.2.171`. To handle this situation, CML allows you to define parameters and filters. A parameter represents a placeholder whose value is extracted from the incoming message. In this example, the parameter is used to extract the software version from the incoming message. Filters specify the bit mask to be used for message
comparisons. In this example, the filter ensures that the incoming message from the hardware device (containing the software version) will match the signal template message (no software version), resulting in the signal being triggered. The \texttt{<sentmessage>} element ensures that the match only succeeds if the incoming message is a response to a \texttt{version.sw} request sent to the hardware device (and not a response to some other request).

**Listing 2. CML to model a signal**

```xml
<signal id="VersionSwReport">
  <message id="VersionSwReportMessage">
    <ascii>ok \r\n\r\n</ascii>
    <parameter type="string">
      <index>3</index>
    </parameter>
    <filter>
      <bytes format="hex">ff,ff,ff</bytes>
    </filter>
    <sentmessage idref="VersionSwGetRequestMessage"/>
  </message>
</signal>
```

A \texttt{<measurement>} element defines a measurement. As mentioned earlier, measurements are implemented using commands and signals. As a result, a measurement definition must reference the command definitions that model the reading and writing of the property and the signal definition that models the hardware device response to the read command. **Listing 3** shows the CML required to model the \texttt{version.sw} property as a read-only measurement. Measurements that are mutable include a \texttt{<writecommand>} element.

**Listing 3. CML to model a measurement**

```xml
<measurement id="VersionSw">
  <readcommand idref="VersionSwGetRequest"/>
  <signal idref="VersionSwReport"/>
</measurement>
```

**More on modeling messages in CML**

As you’ve seen, messages are used when modeling commands and signals. Messages are the basic unit of Device Kit modeling; they are the interface between the hardware device and the Device Kit. Modeling messages is crucial to developing Device Infrastructure device adapters and agents. We’ll give a quick overview of modeling messages. Refer to the Device Developer Device Kit help documentation and the examples provided with the RFID Tracking Kit for more information.

There are two primary types of messages: \textit{concrete} and \textit{template}. Concrete messages are the actual bytes that are sent to or received from a hardware device.
Template messages are patterns that are used to create concrete messages (encoding), extract data from concrete messages (decoding), or match against concrete messages. Concrete messages may be specified in CML and are also produced by (and received by) transports. Template messages are always modeled in CML.

We've already seen one concrete message: the Sirit INfinity version.sw request. When modeling a concrete message in CML, you must include all the bytes of the message. In the case of the VersionSwGetRequest command (see Listing 1), the message includes the \r\n required to end the command by the Sirit INfinity 510 reader protocol.

Template messages are more complex: they include parameters or filters. We've seen one template message: the message used to trigger the VersionSwReport signal (see Listing 2). In that case, the message included a parameter (representing the software version), a filter (to determine if a given concrete message matched the template and should cause the signal to be triggered), and a sent message to ensure that the signal is triggered only for messages that are a response to a version.sw request.

One use of template messages is encoding, or creating a concrete message using parameter data. Encoding is done under the covers when an application executes a command control with a parameter. Listing 4 shows the CML used to model the setup.operating_mode request of the Sirit INfinity 510 reader. The request is used to start (or stop) reading tags. The valid values that can follow the = in the message are standby, polled, and autonomous (the different modes of the reader).

### Listing 4. CML to model a command with a parameter

```xml
<command id="SetupOperatingModeSetRequest">
  <message id="SetupOperatingModeSetRequestMessage">
    <ascii>setup.operating_mode = \r\n</ascii>
    <parameter type="string">
      <insert/>
      <index>23</index>
    </parameter>
    <filter>
      <bytes format="hex">
        ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,
        ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff
      </bytes>
    </filter>
  </message>
</command>
```

You can use a template message to create a concrete message by providing a value for the parameter. The bytes of the data will be inserted starting at index 23.
(0-based) of the concrete message. The `<insert/>` element says to insert the bytes, rather than replace bytes from the template (in that case, a `<length>` or `<size>` element would have to be specified). If an application executes Java code like `commandControl.execute("autonomous")`, then the concrete message `setup.operating_mode = autonomous` is sent to the reader.

Parameters may be one of several types (see the Device Developer Device Kit help for more information). Common ones include `string`, `int`, `byte`, and `nstring` (a string that is used as a numeric value by the application). For the numeric types, the data value used for encoding should be the appropriate Java type (`Integer`, `Byte`, etc). If more than one parameter is defined for a message, you would use an `Object[]` as the data. There are many CML elements available for modeling parameters: bit and byte offsets and lengths, data type conversions, transforms, and so on.

You can also use template messages for decoding, or extracting data from a concrete message. Decoding is done whenever a signal control triggered; the associated data is extracted from the concrete message and made available to any signal listeners. The message defined in Listing 2 can be used to extract the software version number from the response to a `version.sw` request. In that case, the parameter begins at index 3 and continues until the end of the data portion of the concrete message. The definition of the data portion of the message is implemented by the class of which the concrete message is an instance; in this case the data portion is everything up to, but excluding, the `\r\n\r\n` at the end of the message. If a `<length>` or `<size>` element was defined for the parameter, the data would be of the specified size (instead of "the rest of the data"). If more than one parameter is defined for a message, an instance of `Object[]` is returned. There are also ways to return a `List` or `Map` (refer to the Device Kit documentation).

A third use of template messages is for matching, or determining if a concrete message matches a template. Matching is done when trying to determine if an incoming concrete message should trigger a signal. Listing 2 defines a message with a filter that is used to determine whether a signal should be triggered. The filter says that if the first 3 bytes ("ok ") of the template and the concrete messages are the same, the concrete message matches the template. Filters are bit masks and can match parts of bytes, or exclude bits from participating in the match decision by setting filter bits to 0. For instance, a filter of `ff,00,ff` would use the first and third bytes for matching.

Some template messages may specify a sent message to be used during the matching process. In those cases, a match is successful only if the filter returns success and the template-sent message matches the concrete-sent message associated with the concrete message. In the case of Listing 2, for a match to be successful, not only do the first 3 bytes of the concrete message have to match "ok
", but the concrete message that caused the response must match the VersionSwGetRequestMessage defined in Listing 1.

When the Device Kit tools generate Java code from the CML, any message that includes a <parameter>, <filter>, or <sentmessage> element is created as a template; all others are concrete messages.

Using the Device Kit tools

The Device Kit includes wizards for generating Device Kit artifacts. When starting a Device Kit adapter project, you'll use the wizards to generate the initial Device Developer project files and skeleton Java code for both the transport and the device. Once the initial projects are created, you can use the wizards to re-generate the transport and device Java code from the CML files to incorporate message and control definitions added to the files.

When creating a new device adapter, you'll use the following process:

1. Understand the hardware device protocol. Try out commands on the device and verify that the information in the device protocol reference document is accurate.

2. Create the transport and device projects in Device Developer.

3. Develop a first version of a transport that can do basic communication with the hardware device. That means being able to parse incoming bytes into messages and being able to send requests to the hardware device.

4. Model some of the easiest device controls (commands, signals, and measurements). Verify that the device component and transport are working properly together.

5. Add more advanced processing to the transport, such as heartbeats, asynchronous event handling, security, the startup conversation, and so on.

6. Model more of the device controls needed to support the device agent.

7. Iterate between implementing the device agent, modeling the device controls needed to support the agent, and implementing the transport features required.

Next, we'll walk through the initial steps to create the transport and device projects in
Device Developer using the Device Kit tools. The rest of the steps will be described in the later articles in this series.

Create the transport project

Use the Transport Creation wizard, as shown in Figure 7, to create the initial transport Java project, transport classes and transport CML file. Open the wizard in Device Developer by selecting **File => New => Other**, and selecting **Device Kit => Transport**.

**Figure 7. Transport Creation wizard**
Figure 8 shows the SimpleTransport project generated by the transport wizard. The project includes the Java classes `SimpleTransport` (see Figure 9) and `SimpleTransportService` and the CML file `SimpleTransport.cml`.

Figure 8. SimpleTransport project
Figure 9. SimpleTransport class
The generated transport class, SimpleTransport, includes default method implementations for receiving bytes from and sending messages to the hardware device:

```java
protected int processInput(final byte[] bytes, final int length) throws Exception {
    return super.processInput(bytes, length);
}

protected void write(final MessageService message) throws Exception {
    super.write(message);
}
```

As you can see, the methods don't do anything other than call the same methods in their superclass. You will need to re-implement these methods, as well as others, to create a working transport for your hardware device. In the next article in this series, we'll show you how to build a fully functioning transport for the Sirit INfinity 510 reader.

**Create the device Java project, classes, and CML**

Use the Device Creation wizard, as shown in Figure 10, to create the initial device Java project, device classes and device CML file. Open the wizard in Device Developer by selecting `File => New => Other`, then selecting `Device Kit => Device`.

When generating a device, you have the option of specifying the transport service
and implementation classes as shown below. In this example, the service and implementation classes are `SimpleTransportService` and `SimpleTransport` shown earlier.

**Figure 10. Device Creation wizard**

![Device Creation wizard screenshot](image)

*Figure 11* shows the SimpleDevice project generated by the wizard. The project includes the Java classes `SimpleDevice` and `SimpleDeviceService` and the CML file `SimpleDevice.cml`.

*Figure 11* shows the SimpleDevice project generated by the wizard. The project includes the Java classes `SimpleDevice` and `SimpleDeviceService` and the CML file `SimpleDevice.cml`.
Like the generated classes in the SimpleTransport project, there's not a lot of functionality in the SimpleDevice Java classes after the initial generation. Unlike the transport project, however, all the work required to create a functioning device component is done in the CML file. The XML shown in Listing 5 comes from the initial SimpleDevice.cml file generated when the project was created. By adding to this file the CML examples shown earlier and re-generating the device classes from the CML, you can create a device component with the necessary functionality to enable an application to use the INfinity 510 version.sw command.

**Listing 5. CML for SimpleDevice**

```xml
<?xml version="1.0"?>
<!--Licensed Materials - Property of My Company-->
```
Re-generate the SimpleDevice project using the Device Kit tools to add a new class called SimpleDeviceMessages to the project. This class contains fields for the message, parameter and filter definitions added to the CML file along with getter methods for the fields. The SimpleDevice class is also updated to create the necessary control objects for the command, signal and measurement definitions added to the CML file. These updates are shown in Figure 12 and Figure 13. At this point, it's not important to know how the generated code works, only that the device code is "built out" or developed by modeling the device specification in the CML file and using the tools to generate the code. In the third article in this series, we'll give you an in-depth look at how the generated code works.
Figure 12. Updated SimpleDevice project

Figure 13. Updated SimpleDevice Java class

```java
/**
 * Defines the VersionSwGetRequest command.
 */
private final MessageCommand VersionSwGetRequest =
    new MessageCommand(SimpleDeviceService.VersionSwGetRequest,
                        SimpleDeviceMessages.getVersionSwGetRequestMessage());

/**
 * Defines the VersionSwReport signal.
 */
private final DataSignal VersionSwReport =
    new DataSignal(SimpleDeviceService.VersionSwReport,
                    SimpleDeviceMessages.getVersionSwReportMessage());

/**
 * Defines the VersionSw measurement.
 */
private final CommandMeasurement VersionSw =
    new CommandMeasurement(SimpleDeviceService.VersionSw, null, null,
                            null, VersionSwReport, VersionSwGetRequest, null);
```
Conclusion

In this article, you learned about the WebSphere RFID Device Infrastructure component of the WebSphere RFID solution and how you can construct device adapters using the Device Infrastructure Device Kit. You learned about the functions provided by the different adapter components, or layers, and the role CML plays in adapter creation. You also got a brief introduction to the Device Kit tools. The remaining articles in the series will build upon this knowledge. In them, you'll learn how to build a new adapter and agent that you can plug into the WebSphere RFID solution.
Resources

Learn

• Developing a device adapter and agent for the WebSphere RFID Solution, Part 2: Building a transport (developerWorks, 2007): Describes the functions provided by the WebSphere RFID Device Infrastructure component and how to construct device adapters using the WebSphere RFID Device Infrastructure Device Development Kit.

• Websphere RFID Premises Server product overview: A brief overview of the Websphere RFID solution.

• WebSphere RFID Information Center: Product documentation for the Websphere RFID solution.

• Websphere Studio Device Developer help has information for all the RFID TRacking Kit components (after you install the RFID Tracking Kit feature)

• The Sirit Infinity 510 Protocol Reference Guide v1.0 will soon be available at http://www.sirit.com

• Print simple and complex RFID labels with the WebSphere RFID solution (developerWorks, Nov 2006): A guide to printing RFID labels with the Websphere RFID solution.

• Redbook: IBM WebSphere RFID Handbook: A Solution Guide

Get products and technologies

• Download a trial version of WebSphere Studio Device Developer

• Install the RFID Tracking Kit into Device Developer using the Update Manager and defining a site bookmark to http://www-306.ibm.com/software/pervasive/wsdd/updates/571/rfid

About the authors

Karl Freburger

Karl Freburger is an IT Architect with IBM Global Business Services. He works with IBM's Software Group to enable business partners to design solutions that use IBM's RFID components.
Allen Smith

**Allen Smith** is a Senior Software Engineer with the Pervasive Computing Group in Research Triangle Park, North Carolina. He works with business partners to design solutions that use IBM's pervasive middleware. You can contact Allen at allens@us.ibm.com.