iAnywhere®
A SYBASE COMPANY

RFID Anywhere™ Developer's Guide

Part number: DC00257-01-0200-01
Last modified: February 2006

Contents

About This Manual

Subject	This manual introduces common development tasks often undertaken when working with RFID Anywhere. It describes the RFID Anywhere Visual Studio Extension for creating custom business modules, introduces commonly-used interfaces for hardware and component interaction, and outlines how to write Data Protocol Processors (DPPs). Code samples and other useful information is included in this manual.
Audience	This manual is intended for application developers who want to develop applications with RFID Anywhere.
Before you begin	This manual assumes familiarity with RFID, RFID Anywhere, and Visual Studio .NET (C#).

RFID Anywhere documentation

This book is part of the RFID Anywhere documentation set. This section describes the materials in the documentation set and how you can use them.

The RFID Anywhere documentation set

The RFID Anywhere documentation set consists of the following components:

- **RFID Anywhere Getting Started Guide** This manual describes RFID Anywhere, a platform for building RFID solutions. It provides instructions on installing and configuring RFID Anywhere, as well as tutorials that demonstrate how you can use RFID Anywhere to administer and test your RFID network.
- **RFID Anywhere Developer's Guide** This manual describes the RFID Anywhere Visual Studio Extension, and explains how to write custom components, such as business modules for your RFID Anywhere network. It also describes other RFID Anywhere development tasks.
- **RFID Anywhere Help** This collection of HTML files provides help for local services running in RFID Anywhere, and can be viewed from the Administrator Console Properties Manager. These files should be downloaded and extracted to your RFID Anywhere installation directory to make them available when configuring a connector or business module from the Administrator Console.

Visit *http://www.ianywhere.com/developer/product_manuals/rfid_-anywhere/index.html* to obtain the latest version of the RFID Anywhere documentation.

Finding out more and providing feedback

Finding out more

Additional information and resources, including the RFID Anywhere Developer Community, are available at *http://www.ianywhere.com/RFID*.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can email comments and suggestions to the RFID Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to this address, we read all suggestions with interest.

PART I

DEVELOPING WITH RFID ANYWHERE

This part describes how to develop applications for RFID Anywhere

CHAPTER 1

Developing Applications with RFID Anywhere

About this chapter

This chapter introduces the various application development models enabled by RFID Anywhere, and outlines some common development tasks.

Contents

RFID Anywhere development models

RFID Anywhere provides a variety of options for applying business logic, filtering, and grouping to raw RFID data: custom business module development using the Visual Studio Extension, Application Level Events (ALE), Report Engine MP and Web Services.

Since RFID Anywhere manages RFID hardware and integration, developers and integrators can focus on turning raw data into useful information. To enable this transformation of data to information, a wide variety of development options are provided by RFID Anywhere.

For .NET developers, RFID Anywhere includes a Microsoft Visual Studio .NET Extension to create custom business modules. These business modules can process data and integrate with existing enterprise applications. With this extension, RFID Anywhere automatically generates C# code containing all of the necessary environmental references, allowing the developer to focus on core business logic. The framework is tightly integrated with Visual Studio .NET to provide an easy to use and compact environment. This integration allows a developer to use a set of predefined templates and wizards to speed up the development process and to incorporate newly-created components with some standard functionality. Repetitive tasks such as registering for notifications or accessing specific hardware commands and features are automated by wizards. Methods within a business module can also be exposed through web services to allow the methods to be called easily from other applications.

A declarative model for Application Level Events (ALE) allows event cycles to be created and configured without writing code. ALE event cycles generate periodic reports in XML format based on tag activity of EPC tags. Web services can also be used to define or receive event cycle reports.

RFID Anywhere's Report Engine MP functionality offers a declarative model for generating periodic reports in XML format based on tag activity for tags of multiple formats including EPC, ISO, and custom tags. The ability to filter and group raw data before generating and sending a report allows for only the most important information to be sent to enterprise applications.

Common development tasks

This section introduces the common development tasks often undertaken by developers building RFID solutions with RFID Anywhere. A solution may require any number of these tasks to be performed, as showcased through the Inventory Tracker sample application discussed in "Inventory Tracker Demo" [*RFID Anywhere Getting Started Guide*, page 203]. Details about these various tasks are included in this manual, such as:

- Building custom business modules to receive, process and deliver data, and to orchestrate hardware and business logic
- Building or updating web services applications to call web methods running in RFID Anywhere and process the output
- Building or updating enterprise or departmental applications to receive, process and act on output from business modules, ALE reports and multiprotocol reports
- Building a Data Protocol Processor (DPP) to encode and decode custom data

Other tasks that do not involve development, including the defining of ALE event cycle reports and creating simulated data, are discussed in the *RFID Anywhere Getting Started Guide.*

About this document

This document is designed as follow-up material to the *RFID Anywhere Getting Started Guide* and assumes that you have read the *Getting Started Guide* completely.

Note
For more information about the ALE and Report Engine MP development models, or on other RFID Anywhere components such as the RFID Simulator Data Editor that do not require development, see the *RFID Anywhere Getting Started Guide.*

CHAPTER 2

Using the RFID Anywhere Visual Studio Extension to Create Custom Business Modules

About this chapter

This chapter describes the RFID Anywhere Visual Studio Extension and also introduces business modules and their role in an RFID Anywhere network.

Contents

Introduction

The ALE engine and Report Engine MP are provided to generate and receive reports on RFID tag activity. While they are an appropriate choice for some projects, others require a more powerful alternative. By providing custom business modules in addition to ALE and Report Engine MP, RFID Anywhere offers the broadest platform for application design, development, and maintenance.

Custom business modules are edge processing components written with the .NET platform and integrated with RFID Anywhere. They perform the following primary tasks:

- Connecting to RFID controllers to receive events such as RFID tag reads
- Issuing read triggers to hardware to begin collecting information
- Responding to tag events and errors
- Querying and updating secondary sources such as databases and other applications
- Sending processed information to enterprise systems

Developers are given control in each of the above areas to leverage the power of custom code at the edgeware level. This is done through a common API, making the process straightforward.

The included Visual Studio .NET Extension speeds the development process by automatically creating shell code for custom business logic. In addition, the extension builds and deploys these business modules that are automatically detected by RFID Anywhere. Custom business modules allow developers to be vastly more productive and to focus immediately on business logic by performing many often-required tasks automatically.

This strong foundation allows developers to focus on writing business logic immediately rather than designing its supporting structure.

The Visual Studio Extension included with RFID Anywhere provides tools that simplify the development of business modules for your RFID Anywhere network. The tools included in the Visual Studio Extension walk you through the steps for common tasks, including:

- developing custom business modules using a C# template
- adding Setup Projects (used to build *.msi* files that can be deployed to your RFID Anywhere network)
- adding references to controllers to your custom components

- adding OIM notifications to your custom components

Note

The sample source code discussed in this lesson can be found in the Inventory Tracker business module. Open the file *InventoryTracker.csproj*, which is located in the *Inventory Tracker\InventoryTracker\SourceCode* folder of your RFID Anywhere installation directory. Follow along in this file as you read this chapter.

Creating new business modules

The Visual Studio Extension includes templates that enable and simplify the creation of new business modules.

❖ To create a new business module

1. Open Visual Studio .NET.
2. From the File menu, choose New ➤ Project.

   The New Project dialog opens.
3. In the Project Types pane, select Visual C# Projects.
4. In the Templates pane, select RFID Anywhere Business Module.
5. Type a Name and Location for the project and then click OK.

   The New Business Module wizard appears.
6. Select the Export as a Web Service option to allow you to expose any public method you create as a web service. This will create your business module as an RnWebBusinessModule and will allow you to add the RfidNetWebMethod attribute to any public method you wish to expose via web services.

   ☞ For more information on exposing business module methods through web services, see “Exposing business module methods via web services” on page 59.

   Enter values for the Namespace, Description, and Class Name and then click Finish to create the new business module.

   The namespace organizes classes and other types in a single hierarchical structure. It is recommended that you use a unique Namespace for your company.

   The entered description appears in the Administrator Console Home page under the Local Services list. It is recommended that you provide a description that can identify the purpose of the business module.
7. Write the code for your business module.

Attaching to controllers

Custom business modules interact with RFID hardware through controllers. Controllers abstract information from a family of connectors and act as a medium between connectors and business modules. For example, the RfidMPController represents all RFID reader connectors (EPC or ISO). By making reference to controllers (family of connectors) rather than specific connectors, RFID Anywhere business modules are not tied to a particular hardware device.

Each controller type has a corresponding C# class. Once you create a business module, you will most often attach the business module to a controller. You can attach a business module to as many controllers as your application requires.

❖ To attach to a controller

1. Choose New Controller from the RFID Anywhere menu.

   The Add Controller dialog appears.

2. Select the type of controller you want to attach to and then enter the Controller Class Name.

   The Controllers Installed dropdown list includes all the controllers that are implemented in the current version of RFID Anywhere. This list is populated dynamically by querying RFID Anywhere for the list of controllers.

3. Click Finish.

   An instance of the controller is added to your code, as well as the code to start and stop the controller.

In the case of an RFID reader, for example, an instance of the controller class is declared as follows:

```
// Declare a new reader controller and set it to null
RfidMPController mrfidController = null;
```

The RFID Anywhere toolbar in Visual Studio .NET can be used to auto-generate declarations for various kinds of controllers. Once a developer has declared instances of the appropriate controller classes and any supporting variables, they can be used to interact with the rest of the RFID Anywhere system. This is done in two key methods: the Start method, and the Stop method, explained below.

☞ For more information on controllers, see “Understanding RFID Anywhere Controllers” on page 31.

The Start method

RFID Anywhere calls the Start method—`public override void Start`—when the business module begins execution. The Visual Studio .NET extension automatically creates an outline of the method, so developers need only fill in the implementation details.

Two important tasks are normally accomplished here. First, controller references must be assigned to actual controllers and then they must be attached to the business module. For the RFID reader controller example, this is accomplished as follows:

```
// Initialize the controller
mrfidController = ( RfidMPController )
      ServiceFactory.GetService( typeof(RfidMPController) );

// Connect the controller to this business module
ConnectToController( mrfidController, this );
```

Once the controller has been attached to the business module, the IssueReadTrigger method can be called to instruct the RFID reader to begin reading tags. This can also be accomplished from within the Start method:

```
// Instruct the controller to begin reading until
         StopReadTrigger
mrfidController.IssueReadTrigger( null );
```

The controller continues to perform reads until StopReadTrigger has been called once for each corresponding IssueReadTrigger. A null argument is passed in the above example, which sends the command to all readers configured within the RFID Anywhere instance. If the string name of a reader from the controllers' Source List property is provided instead, the command is only sent to that reader. This can be used to achieve more granular control over reader operation.

> **Note**
> If the business module does not co-ordinate readers or start and stop readers from reading (for example, if the Read on Start property of a reader connector is set to True), read triggers do not need to be used in the business module. However, the use of read triggers is recommended to ensure that readers are only reading when necessary, enabling control of interference and reducing reader power consumption.

By default the controller connects to all the sources that belong to it. However, you can specify a string array of the sources you want to connect. For example, you can add the following code to connect only to the sources you want.

```
string[] Readers = {"RfidSimulator1", "RfidSimulator2"};
if( mrfidController != null )
{
 ConnectToController( mrfidController, this, Readers );
}
```

The Stop method

The Stop method—`public override void Stop`—is called when the business module shuts down. Read triggers must be stopped and controllers disconnected during this process. Using the RFID reader controller example from above, this is accomplished as follows:

```
// Stop reading
mrfidController.StopReadTrigger( null );

// Disconnect the controller from the business module
DisconnectFromController( mrfidController );
```

Once again, the null parameter for StopReadTrigger instructs the controller to send the message to all readers. A particular reader's string name can be specified in place of the null.

Note
You do not have to specify the string array of the sources when the DisconnectFromController method is called.

Handling tag events

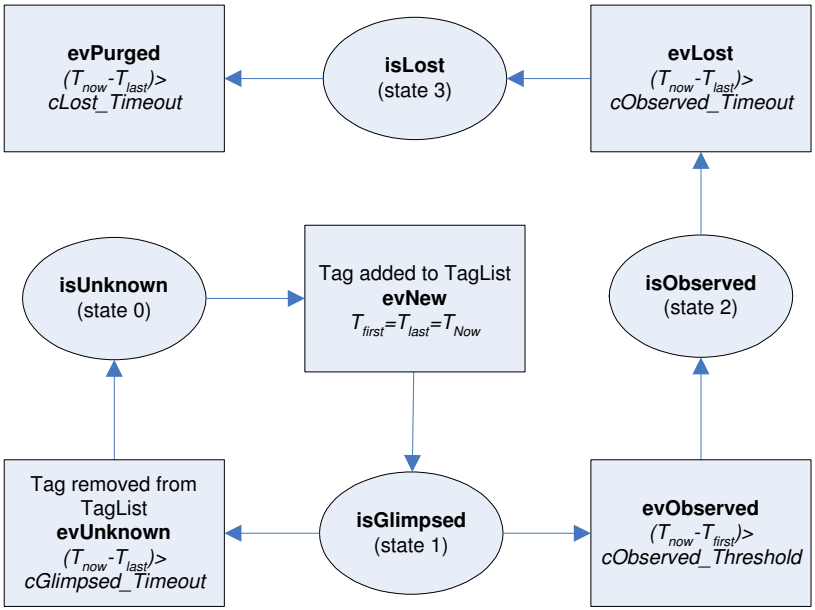
RFID Anywhere uses a set of four **knowledge states** as the basis for tag activity: isUnknown, isGlimpsed, isObserved, and isLost. Movement between states defines an event.

Conceptually, all tags begin in the isUnknown state and generate events as they move from one state to the next. When one or more readers first detect a tag, an evNew event is generated and the tag moves into the isGlimpsed state. If the tag remains visible for a configurable threshold period, it moves to the isObserved state, generating an evObserved event. Otherwise, if the tag disappears for longer than a specified timeout, it moves back to the unknown state, generating an evUnknown event.

Hardware controllers maintain lists of tags present at each reader. If a tag moves into the isGlimpsed state, it is added to the corresponding list. Similarly, if it moves into the isUnknown state, it is removed.

The evObserved event allows applications to discriminate between tags that enter the field only briefly, and tags that remain for a number of consecutive read cycles. The former could be spurious, such as tags outside of a reader's intended monitoring area. Tags that successfully reach the isObserved state stay there until the observed timeout is exceeded, at which point an evLost event is generated while the tag moves to the isLost state. Tags in this state are eventually purged, generating an evPurged event.

Transitions between states are controlled by a set of configurable thresholds that are compared to the difference between the current time, T_{now}, the time the reader first saw the tag T_{first}, or the time the reader most recently saw the tag T_{last}. These smoothing thresholds are configured at each reader connector.



☞ For more information on the smoothing properties exposed by hardware connectors, see the hardware connector's help file.

RFID Anywhere's custom business modules handle these events with the OnRnEvent method, an outline of which is automatically created by the Visual Studio .NET Extension. Developers only need to write logic to process tag events as they arrive. The smoothing parameters to define when these events are generated are configured in the properties of the reader connector.

RFID Anywhere passes a list of recent events to OnRnEvent through an RnEventArgs array. The example below iterates through the list, first looking for RFID-related elements—RFID Anywhere can also handle bar codes, proximity sensors, and other hardware—and then handling any instances of evObserved and evLost. This structure could be expanded to look for events from non-RFID hardware, other events such as evNew and evPurged, or external data sources.

```
public void OnRnEvent( RnEventArgs[] args)
{
 // Iterate through available events
 foreach( RnEventArgs arg in args )
 {
  // Identify RFID events
  if( arg is RfidMPEventArgs )
  {
   // Typecast the generic event to an RFID event
   RfidMPEventArgs rfidEvent = ( RfidMPEventArgs )arg;
```

```
      if( rfidEvents.EventType == TagEventType.Observed )
     {
      // Handle observed tag
           }
     else if( rfidEvents.EventType == TagEventType.Lost )
     {
      // Handle lost tag
     }
     else
     {
      // Handle other events here
     }
    }
    else
    {
     // Handle non-RFID events here
    }
   }
  }
```

Developers can use the RfidMPEventArgs class to retrieve a great deal of information about RFID events. The above example uses only the EventType property. Other important properties exposed by the class include:

- **Source** returns the reader name that detected the event.
- **Tag** returns an RFIDTag object that contains all values of a tag. If the tag is EPC, then the RFIDTag object will contain the entire tag ID. If the tag is ISO, then the RFIDTag object will contain the ID, AFI and Data fields.
- **TagID** actual value stored on the tag. You can directly call this method to access a string value of an EPC tag.
- **TagType** gets the RFID tag type—EPC, ISO180006B, and so on.
- **Time** returns the date and time of the event.

There are also EventArgs classes for proximity sensors, barcode readers, and other hardware. For simplicity and clarity, RFID hardware was used in this example.

☞ For more information on handling events from the various controllers, see "Events from controllers" on page 34, and the sections in "Understanding RFID Anywhere Controllers" on page 31 for each hardware family.

Working with OIM properties

The Object Information Model (OIM) is a set of custom attributes that RFID Anywhere uses to expose an object and its properties through the RFID Anywhere Administrator Console.

OIM properties simply store and retrieve developer-defined properties for use by the custom business module. These properties have get and set methods. OIM properties provide run-time properties to business modules, meaning that logic inside a business module can access the property values without re-compilation. The property values are serialized between starts and stops of the business module.

There are many types of OIM properties. These pre-built property types can be found in the assembly *iAnywhere.RfidNet.OIM.dll.* That assembly is automatically added as a reference to all business modules.

One simple property to use in a business module is the StringProperty. The StringPropertyAttribute class in *iAnywhere.RfidNet.OIM.dll* implements this attribute. The constructor for this class takes a string parameter. This can be seen by using the Object Browser in Visual Studio.

The following code sample demonstrates how to use the StringProperty:

```
// declare a private variable
private string myString = "Test";

// Specify the attribute being used
// Note that the OIM class StringPropertyAttribute is referenced
// as [StringProperty].
// The StringPropertyAttribute has a string constructor and is
// used as a label to identify the attribute.

[StringProperty("StringProperty")]

//Declare public variable and provide get and set methods
public string propertyString
{
 get{ return myString;}
 set{ myString = value;}
}

// Now, the value of this property set at runtime can be
         referred
// to in code using the private variable myString.
```

Examples of various property declarations are discussed below.

ArrayList properties

Array list properties are used to specify multiple entries, each on their own line when editing the property.

```
[ArrayListProperty( "ItemList", typeof(System.String) )]
```

Enumeration properties

Enumeration properties are used to expose a dropdown list of options to the administrator. Here, Locations represents a previously-created object of type EnumType.

```
[EnumProperty( "Location", typeof(Locations) )]
```

Stream properties

Stream properties are used to allow the administrator to browse to a file. The file stream is then accessible to the business module.

```
[StreamProperty("Stream Property-Enter file: ")]
```

Boolean properties

Boolean properties expose a True or False dropdown to the administrator.

```
[BoolProperty("SomeBooleanProperty")]
```

Password properties

Password properties are used to expose a text field to enter a password value which will be replaced with “*****” when they appear in the Properties Manager.

```
[PasswordProperty("DatabasePassword")]
```

Hyperlink properties

The hyperlink property allows a developer to specify a link to a help file for their business module. Specifying this property exposes a clickable help icon in the Properties Manager.

```
[HyperLinkProperty("Help")]
  public String Help
    {
      get { return "docs/MyBusModHelpFile.html";}
    }
```

☞ For more information about adding help files to a business module, see "Adding a Help property" on page 103.

Object properties

Object properties enable the grouping and nesting of related properties. The following example will expose a collapsible property named Nested Object Props, under which are single properties as defined in the SingleObjectProps object passed to the constructor.

```
[ObjectProperty("Nested Object
        Props",typeof(SingleObjectProps))]
```

Object array properties

Object array properties allow administrators to select a configurable number of objects to configure, and once that number is chosen, nested properties are exposed for configuration. The example below exposes a dropdown property called NumberOfObjects, with available values 0 to 10. Once a number (*n*) is selected, nested properties MultipleObjects(1) through MultipleObjects(*n*) are created, each with their own nested properties as defined in the MultipleObjectProps object passed to the constructor.

```
[ObjectArrayProperty("NumberOfObjects",
        typeof(MultipleObjectProps), "MultipleObjects", 10)]
```

Making properties invisible

Invisible OIM properties are not visible from the Administrator Console, but are still serialized to the disk for access across multiple restarts of the business module. A common usage of this type of OIM property is to identify whether or not the business module has been run before. To make an OIM property invisible, add the InvisibleOIMProperty attribute after the OIM attribute reference.

```
[BoolProperty("FirstTimeExecuted")]
[InvisibleOIMProperty()]
```

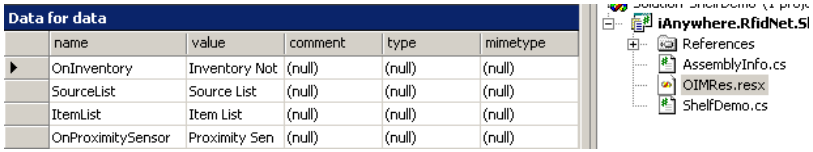
Making properties read-only

To make a property read-only, only implement the get method.

```
private string myStringReadonly = "Readonly";
[StringProperty( "StringPropertyReadonly")]
public string propertyStringReadonly
{
   get{ return myStringReadonly; }
}
```

Using resource files for OIM properties

OIM tables allow developers to provide descriptive names for property descriptions in the RFID Anywhere Administrator Console. The mappings are defined through the *OIMRes.resx* file, as shown in the following screenshot.



Data for data

	name	value	comment	type	mimetype
▶	OnInventory	Inventory Not	(null)	(null)	(null)
	SourceList	Source List	(null)	(null)	(null)
	ItemList	Item List	(null)	(null)	(null)
	OnProximitySensor	Proximity Sen	(null)	(null)	(null)

In the resource file, the developer creates a relationship between the internal label (in the name field) and a more descriptive label (in the value column). In the example above, the OIM property named SourceList appears in the business module Properties Manager as Source List.

Order of properties in Administrator Console

OIM properties appear in the Administrator Console ordered alphabetically by their internal property name. If a particular display order is desired, simply ensure that the internal properties are named accordingly. For example, a developer could prepend numbers to the property names. Since resource files are used to map internal property names to display names for each OIM property, they can easily hide internal naming conventions from administrators.

Sending output through messaging connectors with OIM Notifications

In addition to the OIM properties discussed above, OIM Notifications are special attributes used to allow a business module to connect, at run-time, to one or more messaging connectors to deliver the business module's output. OIM Notifications allow business modules to send data to enterprise systems without needing to know exactly what messaging system will be used.

❖ To add a new OIM notification property

1. From the RFID Anywhere menu, choose New OIM Notification.

   The New Notification dialog opens.

2. Choose a class to add the OIM Notification to from the Class dropdown list.

   The Class dropdown list contains all the classes that are found in the project. This list is populated dynamically.

3. Enter a label to identify the OIM Notification in the Notification Name field.

4. Click Finish.

   A new property that has an attribute associated with it is added to the code.

   ☞ For information about viewing properties in the Administrator Console, see "Using the Properties Manager" [*RFID Anywhere Getting Started Guide*, page 38].

After following the steps in the wizard and entering OnNotification as the Notification Name, the following code is generated in the class file:

```
private RnNotification mOnNotification = new RnNotification();
[NotificationProperty("OnNotification")]
public System.Collections.ArrayList OnNotification
{
 get
   {
    return mOnNotification.TargetList;
   }
   set
   {
    mOnNotification.TargetList = value;
 }
}
```

Note that the NotificationPropertyAttribute is an RnNotification object. Output from the business module can be sent in XML format through the

messaging connectors listed in the business module's OnNotification property by using the RnNotification object's FireXML() method:

```
mOnNotification.FireXML();
```

An example of using the FireXML() method can be found in the source code for the Inventory Tracker business module. The property used by the Inventory Tracker module for OIM Notifications is called Subscribers.

Managing errors successfully

The RFID Anywhere Visual Studio .NET Extension automatically generates another method, OnErrorEvent, for error handling. A single array parameter, RnErrorEventArgs[], provides information about errors including the error message, the hardware from which the error originated, and its date and time. The following example shows how this data might be written to a log.

```
public void OnErrorEvent(RnErrorEventArgs[] args)
{
 foreach(RnErrorEventArgs arg in args)
 {
  TraceLog.WriteError( "Error received: " + arg.Message );
      TraceLog.WriteError( "Error source: " + arg.Source);
      TraceLog.WriteError( "Error date and time: " + arg.Time);
 }
}
```

☞ For more information on handling errors from controllers, see “Errors from controllers” on page 35.

Deploying business modules

There are two alternatives for deployment when building a business module: you can build the project so that the business module is available in the Administrator Console on the computer where the project is built, or you can build the business module and deploy it (called provisioning) to other computers on your RFID Anywhere network. Packages built for provisioning cannot be used on the same computer on which they were built.

Note
When you build a package for provisioning, the business module cannot be used on the same computer as it was built. It is only available to be used on other computers.

❖ To build a solution for same-machine use

1. Write your business module in Visual Studio .NET 2003.
2. Build the solution by choosing Build ➤ Build Solution.
3. Restart the RFID Anywhere service.

   You must restart the RFID Anywhere service before you can view the new business module in the Administrator Console. To restart the service:

   - From the Start menu, choose Programs ➤ RFID Anywhere ➤ Stop Service.

     The RFID Anywhere Service stops.
   - From the Start menu, choose Programs ➤ RFID Anywhere ➤ Start Service.
   - The RFID Anywhere Service starts.
4. Open the Administrator Console. From the Start menu, choose Programs ➤ RFID Anywhere ➤ Administrator Console.

   Your new business module appears in the Local Services List.

Note
Each time you build a new version of a business module, you must restart the RFID Anywhere service. Otherwise, the previous version of the business module remains loaded in the Administrator Console.

❖ To build a solution for deployment to other computers

1. Write your business module in Visual Studio .NET 2003.
2. In the Solution Explorer, select the *.cs* file for your project.
3. From the RFID Anywhere menu, choose New Setup Project.

   The Setup dialog appears.
4. Enter the Setup Name and Projects for your solution, and then click Finish.

   The information for building a deployment package is generated.
5. From the Build menu, choose Build *yoursetup*.

   When the solution builds successfully, a *.msi* package is created that can be deployed to Managed Systems on your RFID Anywhere network.
6. Deploy the business module to the Managed Systems in your RFID Anywhere network.

   ☞ For information about deploying business modules and other custom business modules, see "Deploying packages" [*RFID Anywhere Getting Started Guide*, page 47].

Uninstalling business modules

If you deployed your business module from Visual Studio for same-machine use, then you can to uninstall it by deleting the dll file from the *bms* folder. If you installed your business module using Provisioning for deployment to other computers, then you can use Provisioning to uninstall it.

❖ To uninstall your business module when deployed from Visual Studio

1. From the Start menu, choose Start ➤ Programs ➤ RFID Anywhere ➤ Stop Service.

   The RFID Anywhere service stops.

2. From Visual Studio, open the RFID Anywhere business module you want to uninstall.

3. In the source code, find the 32 digit hexadecimal ID of your business module below the namespace declaration. For example, the Inventory Tracker business module introduced in the *RFID Anywhere Getting Started Guide* has the following ID:

   ```
   namespace inventoryTracker
   {
      /// <summary>
      /// </summary>
      [RnWebBusinessModule("AA6B44F6-9DB6-4961-9DC1-
            D4D2F6788D0C", "Inventory Tracker 3.0" , null)]
   ```

   Make a note of the Business Module ID.

4. Navigate to the *bms* folder located in the RFID Anywhere installation directory. The default location is *C:\Program Files\Sybase\RFID Anywhere\bms.*

5. Right-click the folder that represents your business module ID and choose Delete. Click Yes to confirm the deletion.

   Note
   This procedure can be used no matter which method you used to deploy your business module. When Provisioning is used to uninstall, it performs these steps.

❖ To uninstall your business module using Provisioning

1. Choose Start ➤ Programs ➤ RFID Anywhere ➤ Administrator Console.

   The Administrator Console opens.

2. At the Login page, type your Windows login name and password for the User Name and Password fields, respectively. Click Log On.

   The Home page opens.

3. Click the Provisioning tab in the Administrator Console.

4. Select the business module in the Packages list.

5. Click Uninstall in the Packages menu.

   The package is removed from the list.

6. It is important to note that when a package is uninstalled, the RFID Anywhere service gets restarted. Once the RFID Anywhere service is fully restarted, click Refresh in order to view the updated list.

   The status of the package changes to Stored.

7. Click the Home tab. If the business module is still present under Local Services, then click Refresh to update the list.

When a property of a business module (or a connector) is modified from the Administrator Console Properties Manager, a *.rn* file is created. The *.rn* file stores the values of properties exposed in the Properties Manager and is located in the *data\[GUID]* folder of your RFID Anywhere root directory. This file will not get deleted when the business module is uninstalled.

After uninstalling a business module, if you need to deploy it again, its properties (in the Administrator Console) contain the values you set before. The newly-deployed business module retrieves the property values from the old *.rn* file. If you do not want this to happen, then you must delete the old *.rn* file before deploying the business module.

❖ To clear your business module properties

1. Close the Administrator Console.

2. From the Start menu, choose Start ➤ Programs ➤ RFID Anywhere ➤ Stop Service.

   The RFID Anywhere service stops.

3. From Visual Studio, open your RFID Anywhere business module.

4. In the source code of your business module, find the 32 digit hexadecimal GUID of your business module below the namespace declaration. For example, the Inventory Tracker business module introduced in the RFID Anywhere Getting Started Guide has the following GUID (highlighted):

```
namespace inventoryTracker
{
   /// <summary>
   /// </summary>
   [RnWebBusinessModule("AA6B44F6-9DB6-4961-9DC1-
         D4D2F6788D0C", "Inventory Tracker 3.0" , null)]
```

Make a note of the Business Module GUID.

5. Go to the *data\[GUID]* folder located in the RFID Anywhere installation directory. For example, the default location of the Inventory Tracker business module GUID folder is *C:\Program Files\Sybase\RFID Anywhere\data\AA6B44F6-9DB6-4961-9DC1-D4D2F6788D0C*.
6. Right-click *yourbusinessmodule.rn* and choose Delete. Click Yes to confirm the deletion.

You have now cleared the values of your business module properties that appear in the Administrator Console Properties Manager.

Summary

RFID Anywhere custom business modules are an easy-to-use and powerful method for analyzing, filtering, and managing tag data at the edge of an RFID network. Visual Studio .NET integration and a straightforward API make the process of developing a custom business module quick and painless and OIM properties make them easy to reconfigure after they have been deployed. RFID Anywhere custom business modules are a unique, full-featured development option for RFID applications.

CHAPTER 3

Understanding RFID Anywhere Controllers

About this chapter

This chapter explores the various hardware family controllers provided by RFID Anywhere. It introduces some of the available functionality for each controller and describes how each controller is commonly used from a custom business module as part of your solution.

Contents

Controller overview

In RFID Anywhere, a hardware **controller** exposes the functionality necessary to support a specific family of devices such as RFID readers, bar code scanners, RFID printers, and sensors. The functionality exposed by a given controller is common to all devices in the family of hardware that the controller supports. For example, the RFID reader connector captures and delivers RFID tag events and exposes triggers to allow readers to be controlled from business logic. Business logic running in RFID Anywhere interacts with controllers as they encapsulate the various hardware connectors installed on a system. This allows data from multiple different pieces of hardware to be aggregated seamlessly and used by a single application.

It is also important to remember that configuration of specific pieces of hardware, and the low-level communication to hardware is handled by hardware connectors installed and configured from the Administrator Console. Controllers load connectors into their memory. For example, the multiprotocol RFID reader controller loads any RFID reader hardware connectors and the RFID MultiProtocol Simulator connector into its memory.

Each controller discovered and started by RFID Anywhere creates and runs in its own application domain for security purposes and to ensure that failure in one controller does not bring down the entire system. Communication between controllers on different systems is enabled through .NET remoting. This allows data aggregation and connector discovery regardless of the connector's physical location in an RFID network. RFID Anywhere's underlying queuing infrastructure is important, as it enables instances of specific controller families to talk to each other easily for discovery and heartbeat purposes.

The hardware controllers provided by RFID Anywhere are:

Multi-protocol RFID reader controller This controller is used to interface with RFID readers of various protocols. By using one controller for all RFID readers, a business module can easily support tag data from different protocols.

Bar code reader controller This controller is used to interface with bar code readers. Since bar codes are often used in conjunction with RFID solutions, it is important to be able to support data from both types of hardware from a single business module.

RFID printer controller This controller interfaces with RFID printers and provides methods used to enable the printing of RFID labels. Printing, or

commissioning, of tags is often part of an enterprise's RFID solution.

Proximity sensor controller Conserving energy by not having RFID readers constantly reading is becoming an important requirement for businesses investing in RFID. By integrating with proximity sensors to detect when items enter a certain location, RFID Anywhere can control when RFID readers scan for tags, and can turn them off when there are no assets to be read.

Programmable Logic Controller (PLC) controller This controller allows business modules to interface with PLCs. These hardware devices are often used to manage business processes and enable control of other devices such as light stacks, energy systems, and so on. Data from PLCs can be used by a business module to help orchestrate hardware such as RFID readers, or business logic running in RFID Anywhere can provide the logic behind a PLC that may not inherently have powerful logic capabilities.

RnController base class functionality

Some of the functionality provided by a controller is inherited from the class **iAnywhere.RfidNet.Core.RnController**.

Useful methods provided by this base class include:

Method	Description
RegisterNotification(string *uri*, string *sEventName***)**	Instructs RFID Anywhere to add a given messaging connector to the Error Notification or On *type* Event property, where *type* is specific to the type of controller. For example, the PLC controller could register for notifications for RnErrorEvent events or PLCEvent events. ***uri*** represents a configured messaging connector. ***sEventName*** is one of 'RnErrorEvent' or an event specific to the type of controller. Returns a boolean value indicating success or failure of the command. Return type: boolean
UnregisterNotification(string *uri*, string *sEventName***)**	Instructs RFID Anywhere to remove a given messaging connector from the Error Notification or On *type* Event property, where *type* is specific to the type of controller. ***uri*** represents a configured messaging connector. ***sEventName*** is one of 'RnErrorEvent' or an event specific to the type of controller. Returns a boolean value indicating success or failure of the command. Return type: boolean

Events from controllers

Events from controllers are passed to the OnRnEvent method of business modules registered to the controller. Each type of hardware extends the base event class **iAnywhere.RfidNet.Core.RnEventArgs**. RnEventArgs have the following properties:

The **ID** property of RnEventArgs contains a unique event ID. ID is type

Int32.

The **Source** property of RnEventArgs contains the connector source that generated the event. Source is type String.

The **Time** property of RnEventArgs contains the time the event was generated. Time is of type DateTime.

Errors from controllers

Errors from controllers are passed to the OnErrorEvent method of business modules registered to the controller. The base event class is **iAnywhere.RfidNet.Core.RnErrorEventArgs**. RnErrorEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source, and Time properties, and has the following additional properties:

The **Message** property of RnErrorEventArgs contains the error message. Message is type String.

Multiprotocol RFID reader controller

The Multiprotocol RFID reader controller exposes the necessary functionality for working with RFID readers of various protocols.

The Multiprotocol RFID reader controller class is **iAnywhere.RfidNet.Rfid.Multiprotocol.RfidMPController**.

This controller is named **Rfid Multiprotocol Controller** in the Add Controller dialog of the Visual Studio Extension.

The methods of this controller are also exposed via Web Services through the **RfidMPModule** business module.

Methods contained in the Multiprotocol RFID reader controller can be called from business modules, or from the RFIDMPModule web service. Useful methods of this controller include:

Method	Description
void **IssueReadTrigger(**String *source***)**	Instructs the pre-configured hardware connectors that are part of the given source to begin reading. This method could be called when the business module initializes to have readers always read. Often, this method is called as the result of an event from a proximity sensor to instruct the reader to begin reading when an asset enters a location. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1.

Method	Description
RfidTagList GetClearTagList(String *source***)**	Instructs RFID Anywhere to return the list of current tags and then clear the internal list of tags for a given source to reset smoothing. This method is similar to calling GetTagList followed by calling ClearTagList. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name* name. For example, MyReader\Antenna1. Returns an RfidTagList object containing the list of current tags. This method cannot be invoked if the connector is executing a ReadTrigger. You must stop the read trigger first. Note that when it is called from web services, this method returns a string containing the list of tags.
RfidTagList ReadIDs(String *source***)**	Instructs RFID Anywhere to return the list of tags currently visible by a given source. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1. Returns an RfidTagList object containing the list of current tags. This method returns the tag list even if there is no read trigger in progress. It starts reading, and then creates and returns the tag list. Note that when it is called from web services, this method returns a string containing the list of tags.

Method	Description
void **StopAllReadTrigger(**String *source***)**	Instructs RFID Anywhere to stop all read triggers for the pre-configured hardware connectors that are part of the given source. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1.
void **ClearTagList(**String *source***)**	Instructs RFID Anywhere to clear the internal list of tags for a given source to reset smoothing. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1. This method cannot be invoked if the connector is executing a ReadTrigger. You must stop the read trigger first.
RfidTagList **GetTagList(**String *source***)**	Instructs RFID Anywhere to return the list of tags currently visible by a given source. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1. Returns an RfidTagList object containing the list of current tags. This method can only work if there is a read trigger in progress. Note that when it is called from web services, this method returns a string containing the list of tags.
void **StopReadTrigger(**String *source***)**	Instructs the pre-configured hardware connectors that are part of the given source to stop reading. ***source*** represents the name of a specific connector and antenna of the form *connector\antenna-name*. For example, MyReader\Antenna1.

Tag events from Multiprotocol RFID reader controller

RnEventArgs from tag events from this controller can be successfully cast as **iAnywhere.RfidNet.Rfid.Multiprotocol.RfidMPEventArgs**. RfidMPEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source, and Time properties, and has the following additional properties:

The **EventType** property of RfidMPEventArgs indicates the type of event. The event type can be one of:

- TagEventType.All
- TagEventType.New
- TagEventType.Observed
- TagEventType.Lost
- TagEventType.Purged

EventType is type TagEventType.

The **Tag** property of RFIDMPEventArgs returns an RFIDTag object that contains all values of a tag. If the tag is EPC, then the RFIDTag object will contain the entire tag ID. If the tag is ISO, then the RFIDTag object will contain the ID, AFI and Data fields.

The **TagID** property of RfidMPEventArgs provides the raw tag ID that generated the event. TagID is type String.

The **TagType** property of RfidMPEventArgs provides the tag type to assist in tag decoding. TagType is type String.

GPIO events from Multiprotocol RFID reader controller

Some RFID readers support general input/output (GPIO) to allow additional devices to be attached directly to the reader. A common usage is to attach a proximity sensor to the reader. Often, GPIO events are used from a business module to instruct the reader to read and look for tags for a given period of time after the GPIO event.

A business module developed for this purpose could have the following code in its Start method:

```
\\ Get the GPIO interface and register to GPIO events
gpio = (IGPIO) controller.GetProtocol(typeof(IGPIO));
if (gpio != null)
{
 TraceLog.WriteVerbose("Enabling GPIO events");
 gpio.SetEvents(readerName);      \\gpio is
         iAnywhere.RfidNet.GPIO.IGPIO
}
else
{
 TraceLog.WriteError("Unable to retrieve GPIO interface");
}
```

The business module could have the following code in its Stop method:

```
\\ Remove from events at Stop method
if (gpio != null)
{
 gpio.RemoveEvents(readerName);
 gpio = null;
}
```

The business module's OnRnEvent method could then look for specific GPIO events and instruct the reader to read for a given period of time.

RnEventArgs from GPIO events from this controller can be successfully cast as **iAnywhere.RfidNet.GPIO.GPIOEventArgs**. GPIOEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source and Time properties, and has the following additional properties:

The **Pin** property of GPIOEventArgs is an object of type iAnywhere.RfidNet.GPIO.GPIOPin.

iAnywhere.RfidNet.GPIO.GPIOPin objects have the following properties that can be used to determine how to act on GPIOEventArgs:

The **Number** property of GPIOPin objects contains a number identifying the GPIO pin. Number is type Int16.

The **Status** property of GPIOPin objects contains the status of the GPIO pin and is one of iAnywhere.RfidNet.GPIO.GPIOPinStatus.On, iAnywhere.RfidNet.GPIO.GPIOPinStatus.Off or iAnywhere.RfidNet.GPIO.GPIOPinStatus.Disabled. Status is type iAnywhere.RfidNet.GPIO.GPIOPinStatus.

The **Time** property of GPIOPin objects contains the time of the event. Time is type DateTime.

The **Type** property of GPIOPin objects contains the type of GPIO pin and is one of iAnywhere.RfidNet.GPIO.GPIOPinType.In or iAnywhere.RfidNet.GPIO.GPIOPinType.Out. Type is type

iAnywhere.RfidNet.GPIO.GPIOPinType.

RFID printer controller

The RFID printer controller exposes the necessary functionality for working with RFID printers.

The RFID printer controller class is **iAnywhere.RfidNet.Printer.PrinterController**.

This controller is named **Printer Controller** in the Add Controller dialog of the Visual Studio Extension.

The methods of this controller are also exposed via web services through the **PrinterModule** business module.

Methods contained in the RFID printer controller can be called from business modules, or from the PrinterModule web service. Useful methods

of this controller include:

Method	Description
void **Load(**String *source*, String [] *labelXMLContents***)**	The Load method allows you to load additional labels into the memory of the printer. Load requires two parameters. The first parameter is the name of the printer connector. The second parameter should contain the XML contents of one or more XML input files. Each file should contain one complete label as generated by the Label Designer.
void **Print(**String *source*, String *labelname*, Hashtable *dynamicfields*)	The Print method formats and generates the printer label. Print accepts four parameters. The first parameter is the name of the printer connector. The second parameter should contain the name of a label that has previously been loaded into the system. The third parameter is a Hashtable that contains the key/value pairs for the dynamic fields of the form. When calling the Print method from web services, the hashtable parameter is replaced with two arraylist parameters, with the first being the keys for the dynamic fields and the second being the respective values. It is acceptable for the Key parameter and Value parameter to be empty, provided the label referenced does not expect any dynamic values.
void **UnLoad(**String *source*, String [] *labelnames***)**	The Unload method allows the user to remove a previously loaded label from the printer. Unload requires two parameters. The first parameter is the name of the printer connector. The second parameter should contain the name of one or more previously loaded printer labels.

RFID printer connectors

RFID printer connectors provide the interface to physical RFID printers and carry out the methods of the controller using a low-level interface to the actual hardware. While each RFID printer connector has its own library and specific functionality, some configuration properties and capabilities are common among all RFID printer connectors regardless of manufacturer or model:

Property	Description
Label List	This property has a value between 1 and 10, indicating the number of labels to pre-load into a the printer's memory. This value defines the number of LabelList sub-properties that define the actual labels. The default value for this property is 1.
LabelList(#) | LabelFileName	For each label to be loaded, a corresponding LabelFileName property is exposed. The LabelFileName property points to a label definition (XML) file created with the Label Designer.
LabelList(#) | LabelName	For each label to be loaded, the LabelName property is extracted from the label definition file. This value is defined in the **Text** property of the label, located in the Appearance set of properties from the Label Designer.
MAC Address	Hardware address to locate and communicate with the physical reader on the network.
IP Address	Network address used to locate and communicate with the physical reader on the network.

Note
The RFID printer controller does not generate events.

Bar code scanner controller

The bar code scanner controller exposes the necessary functionality for working with bar code scanners.

The bar code scanner controller class is *iAnywhere.RfidNet.Barcode.BarcodeController*.

This controller is named **Barcode Reader Controller** in the Add Controller dialog of the Visual Studio Extension.

The methods of this controller are also exposed via web services through the **BarCodeModule** business module.

Methods contained in the bar code scanner controller can be called from business modules, or from the BarCodeModule web service. Useful methods of this controller include:

Method	Description
void **IssueReadTrigger(**String *source***)**	Instructs the pre-configured hardware connectors that are part of the given source to read. ***source*** represents a set of pre-configured hardware connectors defined in the BarCode-Module's Sources property.
BarCodeList **GetClearBar-codeList(**String *source***)**	Instructs RFID Anywhere to return the list of current bar codes and then clears the internal list of bar codes for a given source to reset smoothing. This method is similar to calling GetBarcodeList followed by calling ClearBarcodeList. ***source*** represents a set of pre-configured hardware connectors defined in the BarCode-Module's Source List property. Returns a BarCodeList object containing the list of current bar codes. This method cannot be invoked if the connector is executing a ReadTrigger. You must stop the read trigger first. Note that when it is called from web services, this method returns a string containing the list of bar codes.

Method	Description
string **Read(**String *source***)**	Instructs RFID Anywhere to return the bar code currently visible by a given source. ***source*** represents a set of pre-configured hardware connectors defined in the RfidMP-Module's Sources property. Returns a string object containing the current bar code. This method returns the bar code, even if there is no read trigger in progress. It starts reading, and then returns the bar code.
void **StopAllReadTrigger(**String *source***)**	Instructs RFID Anywhere to stop all read triggers for the pre-configured hardware connectors that are part of the given source. ***source*** represents a set of pre-configured hardware connectors defined in the BarCode-Module's Source List property.
void **ClearBarcodeList(**String *source***)**	Instructs RFID Anywhere to clear the internal list of bar codes for a given source. ***source*** represents a set of pre-configured hardware connectors defined in the BarCodeModule's Source List property. This method cannot be invoked if the connector is executing a ReadTrigger. You must stop the read trigger first.

Method	Description
BarCodeList **GetBarcodeList(**String *source***)**	Instructs RFID Anywhere to return the list of bar codes currently visible by a given source. ***source*** represents a set of pre-configured hardware connectors defined in the BarCodeModule's Source List property. Returns an BarCodeList object containing the list of current tags. This method can only work if there is a read trigger in progress. Note that when it is called from web services, this method returns a string containing the list of tags.
void **StopReadTrigger(**String *source***)**	Instructs the pre-configured hardware connectors that are part of the given source to stop reading. ***source*** represents a set of pre-configured hardware connectors defined in the BarCodeModule's Source List property.

Events from bar code scanner controller

RnEventArgs from this controller can be successfully cast as **iAnywhere.RfidNet.Barcode.BarcodeEventArgs**. BarcodeEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source and Time properties, and has the following additional properties:

The **AIM** property contains the symbology identifier if available. AIM is type String.

The **Barcode** property contains the bar code value. Barcode is type String.

The **DataEncoding** property identifies the type of DataEncoding used to encode the bar code value to assist in processing. DataEncoding is type iAnywhere.RfidNet.Barcode.DataEncoding which is one of None, EPC, or ISO_15693_3.

The **EncodeBase64** property indicates whether or not the bar code is encoded using base64 to assist in processing. EncodeBase64 is type boolean.

The **SymbologyType** property contains the name of the symbology type

returned from the connector. SymbologyType is type String.

Proximity sensor controller

The proximity sensor controller exposes the necessary functionality for working with proximity sensors.

The proximity sensor controller class is **iAnywhere.RfidNet.ProximitySensor.ProximitySensorController**.

This controller is named **Proximity Sensor Controller** in the Add Controller dialog of the Visual Studio Extension.

The methods of this controller are also exposed via web services through the **ProximitySensorModule** business module.

Methods contained in the proximity sensor controller can be called from business modules, or from the ProximitySensorModule web service. This controller does not expose any additional methods aside from the methods in the RnController base class.

Events from Proximity sensor controller

RnEventArgs from this controller can be successfully cast as **iAnywhere.RfidNet.ProximitySensor.ProximityEventArgs**. ProximityEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source and Time properties, and has the following additional properties:

The **Status** property is a boolean value, with True indicating that the proximity sensor was triggered, and False indicating that the proximity sensor stopped being triggered. Status is type String.

PLC controller

The PLC controller exposes the necessary functionality for working with programmable logic controllers.

The PLC controller class is **iAnywhere.RfidNet.Plc.PlcController**.

This controller is named **Plc Controller** in the Add Controller dialog of the Visual Studio Extension.

The methods of this controller are also exposed via web services through the **PlcModule** business module.

Methods contained in the PLC controller can be called from business modules, or from the PlcModule web service. Useful methods of this controller include:

Method	Description
Object **GetValue(**String *name***)**	Instructs the pre-configured hardware connectors to get the value of a the specified resource. ***name*** represents a URI that identifies the point (or resource) and/or a specific property of that resource. For example, GetValue("MyTemperaturePoint.-Measurement"). Returns an object containing the value of the resource.
boolean **SetValue(**String *name*, Object *value***)**	Instructs the pre-configured hardware connectors to set the value of a specific property the specified resource. ***name*** represents a URI that identifies the point (or resource) and/or a specific property of that resource. ***value*** represents the desired value of the resource. For example, SetValue("MyTemperaturePoint.-Measurement","Celsius"). Returns a boolean indicating the success or failure of the method call.
string[] **GetIOPorts(**String *source***)**	Instructs RFID Anywhere to return the list of available ports for I/O communication to the hardware. ***source*** represents a hardware connector defined in the PlcModule's Sources property. Returns a string array object containing the available I/O ports.

Events from PLC controller

RnEventArgs from this controller can be successfully cast as **iAnywhere.RfidNet.Plc.PlcEventArgs** for standard PLC events, or **iAnywhere.RfidNet.Plc.IOPortValueChangedEventArgs** to indicate a change in the resource being monitored by an I/O port.

PlcEventArgs inherits from iAnywhere.RfidNet.Core.RnEventArgs for the ID, Source, and Time properties and has the following additional properties:

The **IOPort** property contains the IOPort that generated the event. IOPort is

type String.

IOPortValueChangedEventArgs inherits from iAnywhere.RfidNet.Plc.PlcEventArgs for the IOPort, ID, Source, and Time property and has the following additional properties:

The **OldValue** property contains the original resource that has just been changed for the I/O port in the IOPort property. OldValue is type Object.

The **ChangedValue** property contains the new resource that has just been changed for the I/O port in the IOPort property. ChangedValue is type Object.

CHAPTER 4

Using ALE programmatically

About this chapter

This chapter builds on the introductory information on RFID Anywhere's ALE implementation discussed in the *RFID Anywhere Getting Started Guide.* It outlines the available ALE methods exposed to business modules and through web services.

Contents

Available ALE methods

ALE methods can be called from business modules, or from the ALEService Web Service. Useful methods include:

Method	Description
void **startTrigger(**string *specName***)**	Instructs a pre-configured event cycle connector to start. ***specName*** represents a pre-configured event cycle connector.
void **undefine(**string *specName***)**	Instructs RFID Anywhere to delete the specified event cycle connector. ***specName*** represents a pre-configured event cycle connector.
String **immediate(**string *ECSpec***)**	Instructs RFID Anywhere to create, run and return the results of a temporary event cycle. ***ECSpec*** the XML representation of the event cycle connector to be created. Returns the generated XML event cycle report.
String[] **getECSpecNames()**	Requests a list of names of all configured event cycles. Returns the list of names of all configured event cycles.
String **getStandardVersion()**	Requests the ALE standard version used by RFID Anywhere. Returns the version number of the ALE standard used by RFID Anywhere.
void **stopTrigger(**string *specName***)**	Instructs a pre-configured event cycle connector to stop. ***specName*** represents a pre-configured event cycle connector.

Method	Description
void **unsubscribe(**string *specName*, string *messagingConnector***)**	Instructs RFID Anywhere to remove the specified messaging connector from the list of subscribers for the given event cycle connector. ***specName*** represents a pre-configured event cycle connector. ***messagingConnector*** represents a pre-configured messaging connector.
String **poll(**string *specName***)**	Instructs RFID Anywhere to run a pre-configured event cycle and return the results. ***specName*** represents a pre-configured event cycle connector. Returns the generated XML event cycle report.
void *subscribe(*string *specName*, string *messagingConnector)*	Instructs RFID Anywhere to add the specified messaging connector to the list of subscribers for the given event cycle connector. ***specName*** represents a pre-configured event cycle connector. ***messagingConnector*** represents a pre-configured messaging connector.
String **getECSpec(**string *specName***)**	Requests the XML representation of the specified event cycle connector. ***specName*** represents a pre-configured event cycle connector. Returns the XML representation of the specified event cycle connector
String **getVendorVersion**	Requests the version of the RFID Anywhere ALEService Module. Returns the version number of the RFID Anywhere ALEService Module.

Method	Description
void **define(**string *specName*, string *ECSpec***)**	Instructs RFID Anywhere to create an event cycle connector with the specified name and properties. ***specName*** represents a pre-configured event cycle connector. ***ECSpec*** the XML representation of the event cycle connector to be created.
String[] **getSubscribers(**string *specName***)**	Requests a list of messaging connectors in the Subscribers property of the given event cycle connector. Returns the list of messaging connectors subscribed to the event cycle connector.

CHAPTER 5

Web Services Demo

About this chapter

This chapter introduces the web services demo and shows how to communicate with RFID Anywhere using web services. It discusses the security parameters needed for authentication.

Contents

Introduction

Web services is a software system designed for interoperability between applications running on different computers. It extends the World Wide Web infrastructure to allow applications to communicate with each other.

Each network service has a unique Web Service Definition Language (WSDL), which is essentially an XML file that contains web services-related information such as its location and a list of methods supported. To access a method supported by a service, an application must generate messages as specified by the WSDL. The messages may need to be enclosed in a SOAP envelope before they are transmitted to the service using HTTP.

The advantages of web services are that they use open standards and protocols, they support any operating system, and they can securely work through corporate firewalls.

☞ To learn more about web services, and related protocols and formats, see the World Wide Web Consortium's (W3C) Web Services Activity (http://www.w3.org/2002/ws/)

With a single line of code, RFID Anywhere allows business modules to expose methods via web services. External applications running on different platforms and located across networks can access these methods to receive rich RFID content. Using the ubiquitous formats and protocols of web services, enterprise applications can securely communicate with RFID Anywhere business modules.

Applications can also modify the state of an RFID Anywhere business module by simply accessing the appropriate web method. For example, developers can expose the Start and Stop methods of a business module via web services. External applications located outside the RFID Anywhere network can then call the Start method to start the business module. Once the required processing is completed, they can call the Stop method to turn off the business module.

This chapter introduces you to RFID Anywhere's Web Services sample application and analyzes the source code. The important topics include passing user credentials, building SOAP requests, and using certificates to encrypt SOAP messages.

Exposing business module methods via web services

Within business modules of type RnWebBusinessModule, developers can expose public methods via web services by placing an `[RfidNetWebMethod]` attribute before any method declaration. Web methods are exposed to other web applications through the SOAP protocol. The following example shows the outline of a web method.

```
[RfidNetWebMethod]
public string SomeWebMethod()
{
 // Implement the web method here
}
```

Web methods allow external applications to request information or modify the state of an RFID Anywhere module at any time. To use RfidNetWebMethod, developers can only use intrinsic data types.

Sample web services application

The web services demo provides an example of how an external application can be used to access a web method. The application accesses the GetCurrentInventory method exposed by the Inventory Tracker business module via web services. GetCurrentInventory returns a string that contains the serial number, product name (SKU), company name and reader name for all the tags that are in the vicinity of the readers.

☞ For instructions on starting the web services application, see "Lesson 5: Using web services to access business module methods" [*RFID Anywhere Getting Started Guide*, page 226].

The following section takes you through the WebServicesApp source code.

Note

It is recommended that as you read this chapter, you open the source code for WebServicesApp and follow along with it.

To view the source code of WebServicesApp, you need to have Visual Studio .NET installed in your computer. Open the Visual Studio C# project file *WebServicesApp.csprog* located in the *Inventory Tracker\WebServicesApp* folder of your RFID Anywhere root directory.

CallGetTagList method

The primary method in the WebServicesApp is the CallGetTagList method. This is the method that gets called when the Get Tags button is clicked. This method declares the InventoryTracker web reference and adds the necessary security information to the SOAP message. It then calls the GetCurrentInventory method that is being exposed by the Inventory Tracker business module. This section takes you through the CallGetTagList method.

Initiate a UsernameToken. The UsernameToken stores the user credentials (username and password) that are entered during runtime.

```
UsernameToken userToken = new UsernameToken( user, password,
        PasswordOption.SendPlainText );
```

Initiate the web reference of the RFID Anywhere component that the application is to interact with. In WebServicesApp, a web reference for the Inventory Tracker business module has been created and named InvTracWS.

☞ For information on adding web reference to your application, see "Adding web service references to an application" on page 63.

The WebServicesApp uses Microsoft Web Services Enhancements (WSE),

which is why the *RAComponent*Wse interface is chosen. It inherits from Microsoft.Web.Services2.WebServicesClientProtocol and contains the WSE-specific features needed.

```
InvTracWS.InventoryTrackerWse serviceProxy = new
        InvTracWS.InventoryTrackerWse();
```

Add the previously-created UsernameToken to the WSE SOAP request.

```
serviceProxy.RequestSoapContext.Security.Tokens.Add( userToken
        );
```

Sign the request by adding the signature element to a security section of the request.

```
serviceProxy.RequestSoapContext.Security.Elements.Add( new
        MessageSignature( userToken ) );
```

The WebServicesApp application provides the user with the option of using X509 Certificate to encrypt the SOAP messages. This is an optional feature. The part of code that pertains to the certificate is discussed in the next section.

Set a time frame in which a SOAP message is valid to prevent reply attacks. In this case, the message is valid for 120 seconds.

```
serviceProxy.RequestSoapContext.Security.Timestamp.TtlInSeconds
       = 120;
```

Invoke the GetCurrentInventory method of the Inventory Tracker business module. Since you are expecting a string to be returned, it will get stored directly in a text box.

```
richTextBoxResults.Text =  serviceProxy.GetCurrentInventory();
           listBoxMessages.Items.Add("Called
        GetCurrentInventory() successfully");
```

Depending on the complexity of your application, you may need more lines of code. WebServicesApp provides you with an example of what is required to access an exposed business module web method. It is important that user credentials are included and signed to a security section of the SOAP request.

Note

To develop your own version of the WebServicesApp or a custom web services application, you need to install the Visual Studio Tools when installing Microsoft Web Services Extensions 2.0 SP3.

Using X509 certificates

For those familiar with X509 certificates, RFID Anywhere provides a certificate for use with web service calls. By encrypting the SOAP messages, X509 certificates can add security to data sent to and from an RFID Anywhere service. This section provides an example of obtaining the RFID Anywhere X509 certificate and using it to encrypt SOAP messages.

❖ To get the X509 certificate

1. Initialize the X.509 certificate token.

```
X509SecurityToken securityToken = null;
```

2. Enable access to the X.509 certificates defined in the local computer.

```
X509CertificateStore store =
 X509CertificateStore.LocalMachineStore(
        X509CertificateStore.MyStore );
   bool open = store.OpenRead();
```

3. Locate the certificates identified as “RFID Anywhere” in the certificate store. This is the certificate that RFID Anywhere installs by default.

```
X509CertificateCollection certs =
store.FindCertificateBySubjectString( "RFID Anywhere" );
```

4. Get a single certificate instance from the certificates obtained.

```
X509Certificate cert = ((X509Certificate) certs[0]);
```

Once the X509 certificate is obtained, you can use it to encode SOAP messages that are transmitted to an RFID Anywhere service.

❖ To encrypt SOAP messages

1. Get the X509 certificate using the steps in the previous procedure.

```
X509SecurityToken encryptionToken = from-previous-procedure
```

2. Add the certificate to the WS-Security header of the request.

```
serviceProxy.RequestSoapContext.Security.Tokens.Add(encrypti
        onToken);
```

3. Encrypt the SOAP request using the certificate.

```
EncryptedData enc = new EncryptedData(encryptionToken);
   serviceProxy.RequestSoapContext.Security.Elements.Add(enc
        );
```

4. Specify that the userToken is encrypted using the certificate.

   The userToken contains the user credentials.

   ```
   serviceProxy.RequestSoapContext.Security.Elements.Add(
   new EncryptedData(encryptionToken,"#" + userToken.Id));
   ```

   Once the SOAP message (including the userToken) is encrypted using the certificate, an RFID Anywhere service method can be invoked the same way as described in “CallGetTagList method” on page 60.

Adding web service references to an application

A reference needs to be added for each web service that your application is to communicate with. WebServicesApp references the Inventory Tracker business module.

❖ To add a web reference to your application

1. From Visual Studio .NET, create a new project.
2. From the Project menu, choose Add Web Reference.

   The Add Web Reference window opens.
3. In the URL field, type the URL of the web service you want to connect to.

   The URL for any RFID Anywhere service will be of the format: **http://***IPAddress*/**RFIDAnywhere/***your-business-module***.asmx**. Since Inventory Tracker is running on the local machine, the *IPAddress* is localhost. To access the Inventory Tracker business module, type `http://localhost/RFIDAnywhere/InventoryTracker.asmx`. Click Go to find the service.

   To quickly obtain the URL for an RFID Anywhere service, click the service hyperlink from the Administrator Console Home page.
4. Enter a web reference name and click Add Reference.

   In the WebServicesApp application, the web reference name entered is **InvTracWS**.

   In your Visual Studio Solution Explorer pane, you will see a new **WebReferences** folder added. The folder contains the InvTracWS web reference name. Visual Studio .NET automatically obtains the Web Service Definition Language (WSDL) and the Discovery (Disco) files.

CHAPTER 6

Creating and Using a Data Protocol Processor (DPP)

About this chapter

This chapter builds on the introductory Data Protocol Processor (DPP) content from "Getting Started with Data Protocol Processors (DPPs)" [*RFID Anywhere Getting Started Guide*, page 95] and provides steps and instructions for creating custom DPPs for use with RFID Anywhere. RFID Anywhere provides a DPP for encoding and decoding a number of EPC encodings, so the focus of custom DPP development is often on tags that provide blocks for user, or custom, data.

Contents

Tag type support

To create a DPP for a specific tag, the underlying tag type must be recognized and supported by RFID Anywhere. This ensures that RFID Anywhere can represent the underlying tag type correctly, and identify the unique components of each type, such as the specific bytes allocated for custom data. Tag data can be organized into multiple sets of bytes or blocks.

Tag and protocol capabilities, interoperability requirements, reader choice and environmental factors often drive tag selection.

Tag types are defined in *iAnywhere.RfidNet.Rfid.Tags*. The tag types currently supported by RFID Anywhere include:

Tag type	Object class
EPC Class 0	iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_-Class0
EPC Class 0 Plus	iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_-Class0Plus
EPC Class 1	iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_-Class1
EPC Class 1 Gen 2	iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_-Class1G2
EPC 1.19	iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_-V119
EM Marin 4222	iAnywhere.RfidNet.Rfid.Tags.-RFIDTagEM4222
Impinj Zumma	iAnywhere.RfidNet.Rfid.Tags.-RFIDTagImpinjZumma
ISO 11784	iAnywhere.RfidNet.Rfid.Tags.RFIDTag11784
ISO 15693	iAnywhere.RfidNet.Rfid.Tags.RFIDTag15693
ISO 18000-6A	iAnywhere.RfidNet.Rfid.Tags.-RFIDTag180006A
ISO 18000-6B	iAnywhere.RfidNet.Rfid.Tags.-RFIDTag180006B
UCODE	iAnywhere.RfidNet.Rfid.Tags.-RFIDUCODETag

Tag type	Object class
UCODE revision 2	iAnywhere.RfidNet.Rfid.Tags.- RFIDUCODETagRev2

iAnywhere can build new tag types and add them to RFID Anywhere using RFID Anywhere's plug-and-play model in the same way new hardware connectors can be deployed. These classes encapsulate one instance of a tag and all of the information required to store, parse and work with the tag data.

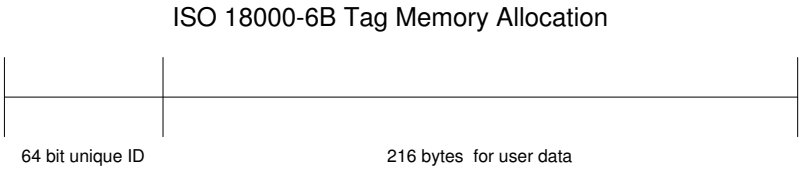
Designing a DPP

As with most development tasks, it is useful to scope out a DPP before writing any code. These steps assume that the business requirements for custom data have been identified, or that you already have custom data encoded onto tags. This chapter goes through the process of designing a DPP to encode and decode tags used to track IT assets inside a company.

Determining the tag type

Begin designing your DPP by identifying the tag type that will be used. This will dictate the amount of memory available to store custom data on the tag. It may be useful to start with a base representation of the tag type being used that can be added to throughout the design process.

The sample IT assets tracking DPP uses an ISO 18000-6B tag that provides a 64-bit unique tag identifier, along with 216 bytes for custom data (of which only a small portion is used fr this sample).



Determining custom data to store

With the flexibility to write custom data to an RFID tag, enterprises have the flexibility to store whatever information they require. Examples include expiration dates on perishable goods, the original manufacturer of a part, or any other data that adds business data to the tracking of an asset.

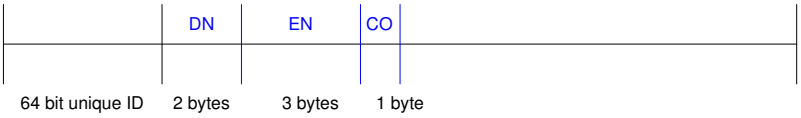
For the sample IT assets tracking DPP, Department Name (DN), Employee Number (EN), and a flag indicating whether the item is company owned (CO) is tracked in the custom data portion of the ISO 18000-6B tag, along with the unique identifier. An ISO 18000-6B tag has three parts where data is stored: ID, Application Family Identifier (AFI), and Data. The ID portion stores the tag identifier and this value is unique for each tag. AFI is used to identify how the information in the Data field is partitioned, and the Data field contains custom data.

Obfuscation, encryption, and other storage strategies

Especially in closed loop systems, enterprises have the flexibility to not only store whatever data they desire, they can also choose *how* to store the data to protect it. For example, an organization looking to write sensitive information to a tag could choose to encrypt the data as part of the DPP's Encode method. The Decode method would use the same algorithm to decrypt the data. By doing this, anyone reading the raw contents of the tag without the proper DPP for decoding would not be able to use the business-level data.

An enterprise could also use bit shifting or any other strategy to obfuscate data during encoding and decoding. For example, the sample DPP's Encode method could store the 2-byte Department Name value over a number of physical bits throughout the custom data memory. The Decode method would need to know this mapping to put the Department Name back together after grabbing the various bits of information. For storage strategies such as this, it is important to scope out the various bits used for storage during this design phase.

For applications that do not require advanced storage strategies, it is still useful to have a clear representation of the data storage specifics. In the sample DPP, the Department Name is stored in the first two bytes of the custom data portion, the Employee Number in the 3 bytes immediately following, and the Company Owned flag (Y/N) in the next byte. Simple encryption is used to secure data. The ID portion of the tag (ISO 18000-6B) is simply stored and retrieved as a string. It is never encrypted since it is just a unique tag identifier. The same principle applies for AFI. The Data portion of the tag, however, needs to be encrypted. It contains valuable information that needs to be protected from outsiders attempting to read the contents of the tag. In the sample DPP, the contents of Data are encoded using ASCII decimal values in the Encode method. The Decode method takes the ASCII decimal values and converts them back to strings.



Building a DPP

Once you have designed your DPP, you can use Visual Studio .NET to create the actual DPP class that will be used by RFID Anywhere.

❖ To create the DPP you designed

1. Open Visual Studio .NET 2003.
2. From the File menu, choose New ➤ Blank Solution.
3. In the list of Visual C# Project templates, click Class Library.
4. In the Name field, enter a name for your DPP. The sample IT assets tracking DPP uses the name DPPITAssets.
5. Browse to the location where you want to save the DPP project.
6. In the Solution Explorer window, right-click the generated *Class1.cs* class and choose Rename.

   Provide a meaningful name for your DPP class. The sample IT assets tracking DPP uses the name *DPPITAssets.cs.*
7. In the *.cs* file, change any references of Class1 to the name of your class.
8. From the Solution Explorer, add references to *iAnywhere.RfidNet.Core*, *iAnywhere.RfidNet.DPP*, and *iAnywhere.RfidNet.Rfid.Tags* by right-clicking the References node in the Solution Explorer window and choosing Add Reference for each one. You must browse to the .NET libraries located in the *bin* directory of your RFID Anywhere installation to locate each file.
9. Add the following lines to the *.cs* file:

   ```
   using iAnywhere.RfidNet.DPP;
   using iAnywhere.RfidNet.Rfid.Tags;
   ```
10. Add attributes to this class so that RFID Anywhere can recognize it as a DPP by adding the following line above the class definition:

    ```
    [DPPModule("ITAssets_DPP", "IT Assets tracking sample DPP")]
    ```

    In this example, ITAssets_DPP is the name used by other RFID Anywhere components to refer to the DPP, while the second parameter is a description.
11. The DPP class you are creating must implement the IDPP interface for base functionality.

To implement the IDDP interface, add : `IDPP` on the class definition line.

12. The namespace of your DPP must be called `iAnywhere.RfidNet.DPP`.

    This makes your DPP class part of the RFID Anywhere DPP namespace. The DPP Manager will be able to recognize your DPP.

13. You must implement the Decode, Encode, GetTypesSupported, and GetSchema methods of the IDPP interface. The documented code sample below provides the implementation of the IT assets tracking DPP.

```
using System;
using System.Data;
using System.Text;
using System.Collections;
using iAnywhere.RfidNet.Core;
using iAnywhere.RfidNet.DPP;
using iAnywhere.RfidNet.DPP.Simulator;
using iAnywhere.RfidNet.Rfid.Tags;
using iAnywhere.RfidNet.Simulator;

namespace iAnywhere.RfidNet.DPP
{
   /// <summary>
   /// This is the IT assets tracking sample DPP.
   /// </summary>
   [DPPModule("ITAssets_DPP", "IT Assets tracking sample
         DPP")]
   public class DPPITAssets : IDPP
   {
      public DPPITAssets()
      {
      }
      #region IDPP Members
```

The Decode method is used to decode tag data. Reader connectors will populate the RnData data object and RawTagDataToDecode will contain values for ID, AFI, and Data.

☞ For information on calling the Decode method from a business module or an external application, see “Using the custom IT assets tracking DPP” on page 83.

The Data field of RawTagDataToDecode contains values in encoded ASCII decimal format. The Decode method decodes these values to strings.

```
public RnInfo Decode(iAnywhere.RfidNet.Core.RnData data)
{
   //Try to cast the data sent to this method as an ISO
         18000-6B tag
   RFIDTag180006B RawTagDataToDecode = data as
         RFIDTag180006B;

   //If unable to cast correctly, throw exception
   if( RawTagDataToDecode == null )
   {
      throw new ArgumentException("Could not cast data as
         RFIDTag180006B type.");
   }
//Create new RnInfo object to hold decoded data
   RnInfo rnInfo = new RnInfo();


 //RawTagDataToDecode will have the "ID", "AFI" and "DATA"
         fields populated. Take those values and assign them
         to columns in the RnInfo object.

   rnInfo.Add(COL_TAG_ID, RawTagDataToDecode.ID);
  rnInfo.Add(COL_AFI, RawTagDataToDecode.AFI);

 //Retreive custom data from the Data field of
         RawTagDataToDecode. Read bytes as per DPP design,
         decode the data and add it to the RnInfo object
 rnInfo.Add(COL_DN,
         Encoding.ASCII.GetString(RawTagDataToDecode.Data.By
         teArray, 0, 2));
 rnInfo.Add(COL_EN,
         Encoding.ASCII.GetString(RawTagDataToDecode.Data.By
         teArray, 2, 3));
 rnInfo.Add(COL_CO,
         Encoding.ASCII.GetString(RawTagDataToDecode.Data.By
         teArray, 5, 1));

 //Provide an ID for the returned schema
 rnInfo.SchemaID =  "ITAssets";

 return rnInfo;
}
public Type[] GetTypesSupported()
{
 //Indicate that only RFIDTag180006B objects will be
         returned by this DPP
 return new Type[] {typeof(RFIDTag180006B)};
}
```

```
//Create variables to use when defining the schema of the
        DPP and in the Encode method
private readonly string COL_TAG_ID = "ID";
private readonly string COL_AFI = "AFI";
private readonly string COL_DN = "DN";
private readonly string COL_EN = "EN";
private readonly string COL_CO = "CO";

//The allotted length (bytes) for the ID, AFI, DN, EN and CO
        values in a tag.
//This is customized as per design of DPP. Note that the
        lengths of ID and AFI may be preset by the type of
        tags you are using. The partitioning of the custom
        data portion is completely dependent on the needs
        of your company. The maximum length (bytes) allowed
        for custom data depends on the tag protocol used.
//For IT assets, we require ID, AFI, DN, EN and CO to have
        5, 1, 2, 3 and 1 characters, respectively.
//We are using the approximation that 1 character = 1 byte.
private readonly int ID_NUMCHAR = 5;
private readonly int AFI_NUMCHAR = 1;
private readonly int DN_NUMCHAR = 2;
private readonly int EN_NUMCHAR = 3;
private readonly int CO_NUMCHAR = 1;
```

The GetSchema method contains the table schema. You declare your table name in this method. The table name for the IT assets tracking DPP is “ITAssets”. When using Report Engine MP to generate reports based on this DPP, you need to specify the table name in the Queries property. For example, if you want the reports to only contain IT assets that are owned by the company, then in the Queries property, type **SELECT * FROM ITAssets WHERE CO = ‘Y’**.

```
public System.Data.DataSet GetSchema()
{
 //Create new DataSet object
 DataSet dset = new DataSet();
 //Create raw DataTable to be added to the DataSet later
 //Table name ITAssets used by Decode method and business
        logic/Report Engine MP queries
 DataTable dtRaw = new DataTable( "ITAssets" );
 //Create DataColumns and add to raw DataTable. Column
        containing unique Tag ID can not be null
 DataColumn colTag = new DataColumn( COL_TAG_ID,
        typeof(string) );
```

```
  colTag.AllowDBNull = false;
  dtRaw.Columns.Add( colTag );
  dtRaw.Columns.Add( new DataColumn(COL_AFI, typeof(string))
          );
  dtRaw.Columns.Add( new DataColumn(COL_DN, typeof(string))
          );
  dtRaw.Columns.Add( new DataColumn(COL_EN, typeof(string))
          );
  dtRaw.Columns.Add( new DataColumn(COL_CO, typeof(string))
          );
  //Add raw DataTable to DataSet before returning DataSet
  dset.Tables.Add( dtRaw );
  return dset;
 }
```

The Encode method encodes the information passed in to the method and then stores it in a new tag object. The RnInfo object that is passed in to the method contains the ID, AFI, Department Name (DN), Employee Number (EN), and company owned flag (CO) as strings in separate columns. The strings are checked to ensure they meet the requirements specified by the DPP. Once checked, the values of the Data field (DN, EN and CO) are appended to a string and then encoded to ASCII decimal format. It is important to note that the ID and AFI fields are not encoded.

The CheckLength method checks whether any of the strings have more characters than are allowed. If one or more of the strings exceed the length, then the encode function is stopped and an error message is sent. The CheckVal method adds padding of “0” to the strings if they contain fewer characters than what are required.

```
public iAnywhere.RfidNet.Core.RnData Encode(RnInfo
        TagInfoToEncode)
{
 //Create an empty RFIDTag180006B object
 RFIDTag180006B NewEncodedTag = new RFIDTag180006B();
 //Check the length of each column and ensure that they
        conform to the space allotted by this DPP
 CheckLength(TagInfoToEncode[COL_TAG_ID].ToString(), ID_
        NUMCHAR, "Tag ID");
 CheckLength(TagInfoToEncode[COL_AFI].ToString(), AFI_
        NUMCHAR, "AFI");
 CheckLength(TagInfoToEncode[COL_DN].ToString(), DN_NUMCHAR,
        "Department Name");
 CheckLength(TagInfoToEncode[COL_EN].ToString(), EN_NUMCHAR,
        "Employee Number");
 CheckLength(TagInfoToEncode[COL_CO].ToString(), CO_NUMCHAR,
        "Company Owned");
```

```
 //Check values of each column and add a padding of "0" if
        necessary
 string tagID = CheckVal(TagInfoToEncode[COL_TAG_
        ID].ToString(), ID_NUMCHAR);
 string tagAFI = CheckVal(TagInfoToEncode[COL_
        AFI].ToString(), AFI_NUMCHAR);
 string tagDN = CheckVal(TagInfoToEncode[COL_DN].ToString(),
        DN_NUMCHAR);
 string tagEN = CheckVal(TagInfoToEncode[COL_EN].ToString(),
        EN_NUMCHAR);
 string tagCO = CheckVal(TagInfoToEncode[COL_CO].ToString(),
        CO_NUMCHAR);

 //Build up string with data from RnInfo object passed to
        method
 //This string will ultimately be the encoded custom data
        portion of the tag
 StringBuilder sb = new StringBuilder();
 //String is built up using one column after the other as
        per DPP design
 sb.Append( tagDN );
 sb.Append( tagEN );
 sb.Append( tagCO );

//Add passed ID to NewEncodedTag in Unique ID portion of tag
 NewEncodedTag.ID = tagID;
 //Add passed AFI to NewEncodedTag in AFI portion of tag
 NewEncodedTag.AFI = Convert.ToByte(tagAFI);
 //Convert the string containing DN, EN and CO to ASCII
        decimals and add to user/custom data portion of tag
 NewEncodedTag.Data = new TagData(
        Encoding.ASCII.GetBytes(sb.ToString()) );
 //Return encoded tag
 return NewEncodedTag;
}

iAnywhere.RfidNet.Core.RnData
        iAnywhere.RfidNet.DPP.IDPP.Encode(RnInfo
        TagInfoToEncode, Type type)
{
 //Only one option since DPP only used for RFIDTag180006B
        tags
 //Could throw error if type passed was not RFIDTag180006B
        (use GetTypesSupported method)
 return Encode(TagInfoToEncode);
}
#endregion
```

The CheckVal, and CheckLength methods are custom methods that apply to the IT assets tracking DPP. They are not required for other DPPs.

The CheckVal method is called by the Encode method. It first converts all letters to uppercase and then checks whether any of the strings contain fewer than required characters. A padding of “0” is added if they contain fewer characters.

```
public string CheckVal (string val, int numchar)
{
   val = val.ToUpper();
   string tempVal = "";
   string pad = "0";
   int numCharsToAdd = 0;
   if (val.Length < numchar)
   {
      numCharsToAdd = numchar - val.Length;
      for (int I = 0; I < numCharsToAdd; I++)
      {
         tempVal = val;
         val = pad + tempVal;
      }
   }
   return val;
}
```

The CheckLength method is called by the Encode method. It checks whether any of the strings have more characters than are allowed. If one or more of the strings exceed the length, then the encode function is stopped and an error message is sent.

```
public void CheckLength(string val, int numchar, string
        description)
{
   if(val.Length == 0)
   {
      throw new ArgumentException("You did not enter a value
         for " + description);
   }
   else if (val.Length > numchar)
   {
      throw new ArgumentException("The value for " +
         description + " is too long. It should only be " +
         numchar + " characters");
   }
}
```

14. You need to make sure the assembly name of your DPP is in the correct format.
    a. From the Solution Explorer in Visual Studio, select your project file or *yourDPP.cs* file.
    b. From the Project menu, choose Properties.
    c. Under Common Properties, select General.
    d. In the Assembly Name field, ensure the value is of the form: **iAnywhere.RfidNet.DPP.*yourDPP***.

       For example, IT assets DPP has the value iAnywhere.RfidNet.DPP.DPPITAssets.

e. In the Default Namespace field, ensure you have the following value: **iAnywhere.RfidNet.DPP**

f. Click Apply to apply the changes and then Click OK to close the window.

15. From the Build menu, choose Configuration Manager.

16. Choose Release from the Active Solution Configuration dropdown list. Click Close to close the window.

17. To build your DPP, choose Build Solution from the Build menu in Visual Studio.

    The DLL of your DPP is saved in your project folder.

Installing a DPP

Once a DPP has been built into a DLL, it must be placed in the *bin\DPP* folder of your RFID Anywhere installation. Once RFID Anywhere is restarted, this DPP is discovered by RFID Anywhere for use with other RFID Anywhere components.

Using a DPP with Report Engine MP

DPPs are used inherently by Report Engine MP. The DPP property of an RFID MultiProtocol Report Definition connector specifies the DPP to use for each type of tag that may be fed into Report Engine MP. For example, a DPP property of **RFIDTagEPC_Class0:RA_DPP** specifies that the RFID Anywhere default DPP should be used to decode any EPC Class 0 tag. To use the IT assets DPP, type **RFIDTag180006B:ITAssets_DPP**. The value RFIDTag180006B specifies the type of tags to report on and since IT assets DPP only reports on ISO 18000-6B RFID tags, this is the only option.

The Queries property of an RFID Multiprotocol Report Definition connector specifies the SQL statements to use to filter and group data to be included in the report. For example, when using the default RFID Anywhere DPP, a Queries property of `SELECT * FROM sgtin96 WHERE CompanyPrefix = 1` will include in its generated report only SGTIN96-encoded tags whose CompanyPrefix field has a value of 1. If the IT assets tracking DPP is used, then a Queries property of `SELECT * FROM ITAssets WHERE CO = 'Y'` will include in its reports only IT assets that are owned by the company.

Using a DPP with custom business modules and external applications

DPPs can be used to encode and decode tags in custom business modules and external applications. This section outlines how to use both the RFID Anywhere default DPP for EPC tags and a custom DPP programmatically.

Using the RFID Anywhere default DPP

This section shows you how to use the RFID Anywhere default DPP to encode and decode EPC tags. It is important to note that the steps illustrated in this section can be applied from an external application or a business module.

Encoding tags

Encoding of RFID tags is performed when tags are written. RFID Anywhere's default DPP allows you to encode tags in a variety of encoding types that are supported by EPC. This section looks at an external application that encodes tags in the GID-96 EPC format. The General Identifier 96-bit encoding (GID-96) consists of three fields: General Manager, Object Class, and Serial Number. The application asks the user to input values for these fields.

☞ For information on the column names (properties) of all EPC tag types supported by the RFID Anywhere default DPP, see "Using the default RFID Anywhere DPP for EPC tags" on page 88.

❖ **To encode GID-96 EPC tags using the RFID Anywhere default DPP and retrieve the encoded tag ID**

1. Open the project in Visual Studio. Under the Project menu, go to Add Reference and the add the following references from the *bin* directory of your RFID Anywhere installation:
   - *iAnywhere.RfidNet.Core.dll*
   - *iAnywhere.RfidNet.DPP.dll*
   - *iAnywhere.RfidNet.Rfid.dll*
   - *iAnywhere.RfidNet.Rfid.Tags.dll*
   - *iAnywhere.RfidNet.DPP.RA.dll**

   *This file is found in the *bin\DPP* folder of your RFID Anywhere installation.
2. Create an IDPP object. If the DPP is null, then use the RFID Anywhere default DPP.

```
IDPP dpp = null;

            if( dpp == null )
            {
               dpp = DPPManager.Load( "RA_DPP" ) as IDPP;
            }
```

3. The RFIDEPC_GID96Info constructor accepts unassigned integers as values for the General Manager, Object Class, and Serial Number. Get values from the user (via textboxes) and assign them to three `uint` variables.

```
uint genMgr = uint.Parse(txtgm.Text);
            uint objClass = uint.Parse(txtoc.Text);
            uint serNo = uint.Parse(txtsn.Text);
```

4. Create a new RFIDEPC_GID96Info object and store the values in it.

```
RFIDEPC_GID96Info Info = new RFIDEPC_GID96Info ( genMgr,
         objClass, serNo );
```

5. Encode the data and store the result in an RnData object.

```
RnData data = dpp.Encode(Info);
```

6. Display the encoded tag ID.

```
txtResults.Text = data.Value;
```

Decoding tags

The Inventory Tracker business module uses a DPP to decode EPC SGTIN-96 tag data before building up output to send to the enterprise application. It gets encoded hexadecimal tag IDs from the two RFID MultiProtocol Simulator connectors. This is the format in which a typical RFID reader sends information. To take advantage of the Electronic Product Code (EPC) and obtain the company prefix and item reference for each tag ID, it is necessary to decode the tag IDs into EPC format. The RFID Anywhere default DPP can be used for this.

The RFID Anywhere default DPP is a DPP that is provided with RFID Anywhere that can be used to encode and decode EPC tags. This section examines how the Inventory Tracker demo uses the default DPP to decode EPC SGTIN-96 tags and retrieve the various tag properties.

Note

It is recommended that as you read this lesson, you open the source code for Inventory Tracker business module and follow alongside.

To view the source code of Inventory Tracker business module, you need to have Visual Studio .NET installed in your computer. From Visual Studio.NET, open *InventoryTracker.csproj* located in the *Inventory Tracker\InventoryTracker\Source* folder of your RFID Anywhere installation directory.

The OnRnEvent method receives new RnEvenArgs from the reader / simulator connector(s). This method is automatically called whenever a new event is received from the readers(s). The RnEventArgs objects are checked whether they are of type RfidMPEventArgs and are then cast as RfidMPEventArgs objects. RfidMPEventArgs object provides access to tag properties such as EventType and Tag ID.

```
public void OnRnEvent(RnEventArgs[] args)
      {
            // If GetCurrentInventory() is being called, then
         skip this method. This is to ensure that the resources
         // are free.
         if (!webMethodActive)
         {
            foreach(RnEventArgs arg in args)
            {
               if(arg is RfidMPEventArgs)
               {
                  RfidMPEventArgs rfidEvent = (RfidMPEventArgs)
         arg;

                                 // Check if new tags were
         observed or existing tags were lost
                  if(rfidEvent.EventType ==
         TagEventType.Observed || rfidEvent.EventType ==
         TagEventType.Lost)
                  {
                     // Send the observed tag's encoded ID to
         the EPCTag class for
                     // storage and decoding
                     EPCTag.DecodeTag(rfidEvent.Tag);
```

The DecodeTag method of Inventory Tracker business module is where the RFIDTag object, which contains tag information such as the tag ID, is passed and then decoded.

❖ **To decode SGTIN-96 EPC tags using RA default DPP and retrieve decoded values**

1. Create an RnInfo object to hold the different elements of the EPC tag. The RnInfo object generates a table for the tag. The columns of the table represent tag attributes.

```
protected RnInfo EPC_SGTIN96;
```

2. Inside DecodeTag method, create an IDPP object. If the DPP is null, then use the RFID Anywhere default DPP.

```
public void DecodeTag(RFIDTag tag)
{
 IDPP dpp = null;
   if( dpp == null )
   {
      dpp = DPPManager.Load( "RA_DPP" ) as IDPP;
   }
```

3. Decode the tag from the encoded hexadecimal value to EPC. Store the decoded value in the RnInfo object.

```
EPC_SGTIN96 = dpp.Decode ( tag );
```

   The decoded EPC tag IDs are now stored in the RnInfo object. Now each property of the tag (Filter, CompanyPrefix, ItemReference, and Serial Number) can be accessed individually. The GetDecodedTagID method of the Inventory Tracker business module retrieves these properties and then concatenates the values to form a string representing the entire EPC tag URI.

4. Create and return the entire EPC tag as a string variable. Get each property of the EPC tag and then concatenate them. Each property refers to a column name in the RnInfo object. The column names are specified when the DPP is created.

```
return EPC_SGTIN96.GetProperty("Filter").ToString() + "." +
         EPC_SGTIN96.GetProperty("CompanyPrefix").ToString()
         + "." + EPC_
         SGTIN96.GetProperty("ItemReference").ToString() +
         "." + EPC_
         SGTIN96.GetProperty("SerialNumber").ToString();
```

☞ For information on the column names (properties) of all EPC tag types supported by the RFID Anywhere default DPP, see “Using the default RFID Anywhere DPP for EPC tags” on page 88.

Using the custom IT assets tracking DPP

This section shows you how to use the IT assets tracking DPP to encode and

decode ISO 18000-6B RFID tags. It is important to note that the steps illustrated in this section can be applied from an external application or a business module.

Encoding

Encoding of RFID tags are performed when tags are written. The IT assets DPP can be used to encode ISO 18000-6B tags in a custom format. This section looks at an external application that encodes tags using the IT assets DPP. The ISO 18000-6B tags consist of three fields: ID, AFI, and Data. The application asks the user to input values for these fields.

❖ To encode ISO 18000-6B tags using IT assets DPP and retrieve encoded tag ID

1. Open the project in Visual Studio. Under the Project menu, go to Add Reference and the add the following references from the *bin* directory of your RFID Anywhere installation:
   - iAnywhere.RfidNet.Core.dll
   - iAnywhere.RfidNet.DPP.dll
   - iAnywhere.RfidNet.Rfid.dll
   - iAnywhere.RfidNet.Rfid.Tags.dll
   - iAnywhere.RfidNet.DPP.DPPITAssets.dll*

   *This file is found in the *bin\DPP* folder of your RFID Anywhere installation.

2. Create an IDPP object. If DPP is null then use ITAssets_DPP.

```
IDPP dpp = null;

            if( dpp == null )
            {
               dpp = DPPManager.Load( "ITAssets_DPP" ) as
         IDPP;
            }
```

3. Create a new RnInfo object.

```
RnInfo info = new RnInfo();
```

4. Create columns in the RnInfo object and populate them with values received from the user (via textboxes). The column names correspond to the DPP table schema.

```
info.Add( "ID", txtID.Text );
            info.Add( "AFI", txtAFI.Text);
            info.Add( "DN", txtDN.Text );
            info.Add( "EN", txtEN.Text );
            info.Add( "CO", txtCO.Text );
```

5. Encode the RnInfo object and store the values in an RFIDTag180006B object.

```
RFIDTag180006B tag = dpp.Encode( info ) as RFIDTag180006B;
```

6. Display the complete encoded tag value. Note that since AFI is stored as a byte, you need to convert it to a string. Data is also stored as a byte array, so we need to loop through the array and convert each element to string.

```
txtResults.Text = tag.ID + tag.AFI.ToString();
            for (int I = 0; I < tag.Data.ByteArray.Length; I
         ++)
            {
               txtResults.Text +=
         tag.Data.ByteArray[i].ToString();
            }
```

Decoding

Decoding of RFID tags needs to be performed every time a reader detects a tag. The IT assets tracking DPP can be used to decode ISO 18000-6B tags. The following procedure looks at a custom business module that decodes tags using the IT assets DPP. The BM subscribes to the RFID multiprotocol reader controller and receives tag events.

Open the business module in Visual Studio. Under the Project menu, go to Add Reference and the add the following references from the *bin* directory of your RFID Anywhere installation:

- iAnywhere.RfidNet.Core.dll
- iAnywhere.RfidNet.DPP.dll
- iAnywhere.RfidNet.Rfid.dll
- iAnywhere.RfidNet.Rfid.Tags.dll
- iAnywhere.RfidNet.Rfid.Multiprotocol.dll
- iAnywhere.RfidNet.DPP.DPPITAssets.dll*

*This file is found in the *bin\DPP* folder of your RFID Anywhere installation.

The OnRnEvent method of the business module receives new RnEvenArgs from the reader connector(s). This method is automatically called whenever a new event is received from the reader(s). The RnEventArgs objects are checked whether they are of type RfidMPEventArgs and are then casted as

RfidMPEventArgs objects. This gives access to the RFIDTag object, which contains tag properties such as ID, AFI and Data.

```
public void OnRnEvent(RnEventArgs[] args)
{
 foreach (RnEventArgs arg in args)
 {
      if (arg is
         iAnywhere.RfidNet.Rfid.Multiprotocol.RfidMPEventArgs)
      {
   RfidMPEventArgs tempArg = (RfidMPEventArgs)arg;
   DecodeTags(tempArg.Tag)
```

The DecodeTags method is a custom method that accepts an RFIDTag object as input, decodes the tag, and returns a string that contains the tag ID, AFI and decoded custom data.

❖ To decode ISO 18000-6B tags using IT assets DPP and retrieve decoded values

1. Inside the DecodeTags method, create an IDPP object. If DPP is null then use ITAssets_DPP.

```
public string DecodeTags (RFIDTag tag)
{
 IDPP dpp = null;
 if( dpp == null )
 {
  dpp = DPPManager.Load( "ITAssets_DPP" ) as IDPP;
 }
```

2. Call the Decode method of the DPP by sending in the RFIDTag object `tag`. Store the decoded values in an RnInfo object.

```
RnInfo info = new RnInfo();
            info = dpp.Decode ( tag );
```

3. Create and return the entire the ID, AFI and the decoded Data portions of the tag as a string variable. Get each property of the tag and then concatenate them. Each property refers to a column name in the RnInfo object. The column names are specified when the DPP is created.

```
return (info.GetProperty("ID").ToString() + " " +
         info.GetProperty("AFI").ToString() + " " +
         info.GetProperty("DN").ToString() + " " +
         info.GetProperty("EN").ToString() + " " +
         info.GetProperty("CO").ToString());
}
```

Using the DPP Manager

The code samples above used the DPP Manager to load the default RFID Anywhere DPP. The DPP Manager provides the architecture and interfaces to load DPPs and to work with DPPs. The DPP Manager assembly is *iAnywhere.RfidNet.DPP.dll*, and it contains the IDPP interface. The DPP Manager maintains a list of available DPP IDs, which it keeps up-to-date using reflection. As discussed above, each DPP class has an attribute that uniquely identifies it, and it is this identifier that the DPP Manager uses to load DPPs.

Common DPP Manager methods used from business modules include the following.

DPPManager.Load(uniqueDPPid)

The Load method takes as a parameter the unique ID of the DPP to load. If no DPP is specified, the default RFID Anywhere DPP for EPC, RA_DPP, is used.

```
//Load the DPP with ID RA_DPP
IDPP dpp = DPPManager.Load("RA_DPP") as IDPP;
//Use the loaded DPP to decode an EPC tag value
RnInfo info = dpp.Decode(new
         iAnywhere.RfidNet.Rfid.Tags.RFIDTagEPC_
         Class1("302400000000204000000000"));
```

DPPManager.ListAllIDs()

The ListAllIDs method takes no parameters, and returns a collection containing the IDs of all of the available DPPs.

```
ICollection allDPP = (ICollection) DPPManager.ListAllIDs();
string[] strAllDPP = new string[allDPP.Count];
allDPP.CopyTo(strAllDPP,0);
foreach (string str in strAllDPP)
{
   TraceLog.WriteVerbose("DPP ID: " + str);
}
```

Using the default RFID Anywhere DPP for EPC tags

RFID Anywhere includes a DPP to enable the encoding and decoding of EPC tags. This DPP, called *iAnywhere.RfidNet.DPP.RA.dll* and found in the *bin\DPP* folder of your RFID Anywhere installation directory, enables the creation of tag objects for a number of EPC encodings, and provides simple access to the various fields of each encoding.

Tag Info classes

The RFID Anywhere EPC DPP exposes a number of classes that extend the base *iAnywhere.RfidNet.DPP.RfidTagInfo* class. These classes represent the various EPC encodings that RFID Anywhere supports with the default DPP. Of course, other Tag Info classes can be created by developers to meet specific business requirements.

These classes, and the available methods and properties of each are outlined in the sections below.

RFIDEPCTagInfo

This class inherits from RfidTagInfo, and exposes properties specific to EPC tags.

The **Encoding** property represents the encoding of the underlying tag data to assist with casting and business logic.

The **Uri** property contains the URI representation of the tag.

RFIDEPC_GIAI64Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the GIAI64 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **giai64**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefixIndex** property represents the CompanyPrefixIndex field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefixIndex**.

The **AssetReference** property represents the AssetReference field of the tag. The column name used in Report Engine MP SQL statements for this field is

AssetReference.

RFIDEPC_GIAI96Info

This class inherits from RFIDEPC_GIAI64Info and exposes properties specific to EPC tags encoded with the GIAI96 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **giai96**.

The **Partition** property represents the Partition field of the tag. The column name used in Report Engine MP SQL statements for this field is **Partition**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefix** property represents the CompanyPrefix field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefix**.

The **AssetReference** property represents the AssetReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **AssetReference**.

RFIDEPC_GID96Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the GID96 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **gid96**.

The **GeneralManagerNumber** property represents the GeneralManagerNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **GeneralManagerNumber**.

The **ObjectClass** property represents the ObjectClass field of the tag. The column name used in Report Engine MP SQL statements for this field is **ObjectClass**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_GRAI64Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the GRAI64 encoding.

The table name used in Report Engine MP SQL statements for tags of this

encoding is **grai64**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefixIndex** property represents the CompanyPrefixIndex field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefixIndex**.

The **AssetType** property represents the AssetType field of the tag. The column name used in Report Engine MP SQL statements for this field is **AssetType**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_GRAI96Info

This class inherits from RFIDEPC_GRAI64Info and exposes properties specific to EPC tags encoded with the GRAI96 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **grai96**.

The **Partition** property represents the Partition field of the tag. The column name used in Report Engine MP SQL statements for this field is **Partition**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefix** property represents the CompanyPrefix field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefix**.

The **AssetType** property represents the AssetType field of the tag. The column name used in Report Engine MP SQL statements for this field is **AssetType**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_SGLN64Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the SGLN64 encoding.

The table name used in Report Engine MP SQL statements for tags of this

encoding is **sgln64**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefixIndex** property represents the CompanyPrefixIndex field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefixIndex**.

The **LocationReference** property represents the LocationReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **LocationReference**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_SGLN96Info

This class inherits from RFIDEPC_SGLN64Info and exposes properties specific to EPC tags encoded with the SGLN96 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **sgln96**.

The **Partition** property represents the Partition field of the tag. The column name used in Report Engine MP SQL statements for this field is **Partition**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefix** property represents the CompanyPrefix field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefix**.

The **LocationReference** property represents the LocationReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **LocationReference**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_SGTIN64Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the SGTIN64 encoding.

The table name used in Report Engine MP SQL statements for tags of this

encoding is **sgtin64**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefixIndex** property represents the CompanyPrefixIndex field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefixIndex**.

The **ItemReference** property represents the ItemReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **ItemReference**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_SGTIN96Info

This class inherits from RFIDEPC_SGTIN64Info and exposes properties specific to EPC tags encoded with the SGTIN96 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **sgtin96**.

The **Partition** property represents the Partition field of the tag. The column name used in Report Engine MP SQL statements for this field is **Partition**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefix** property represents the CompanyPrefix field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefix**.

The **ItemReference** property represents the ItemReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **ItemReference**.

The **SerialNumber** property represents the SerialNumber field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialNumber**.

RFIDEPC_SSCC64Info

This class inherits from RFIDEPCTagInfo and exposes properties specific to EPC tags encoded with the SSCC64 encoding.

The table name used in Report Engine MP SQL statements for tags of this

encoding is **sscc64**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefixIndex** property represents the CompanyPrefixIndex field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefixIndex**.

The **SerialReference** property represents the SerialReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialReference**.

RFIDEPC_SSCC96Info

This class inherits from RFIDEPC_SSCC64Info and exposes properties specific to EPC tags encoded with the SSCC64 encoding.

The table name used in Report Engine MP SQL statements for tags of this encoding is **sscc96**.

The **Partition** property represents the Partition field of the tag. The column name used in Report Engine MP SQL statements for this field is **Partition**.

The **Filter** property represents the Filter field of the tag. The column name used in Report Engine MP SQL statements for this field is **Filter**.

The **CompanyPrefix** property represents the CompanyPrefix field of the tag. The column name used in Report Engine MP SQL statements for this field is **CompanyPrefix**.

The **SerialReference** property represents the SerialReference field of the tag. The column name used in Report Engine MP SQL statements for this field is **SerialReference**.

CHAPTER 7

Code Samples

About this chapter

This chapter provides code samples for a number useful development tasks that are often requirements of applications developed using the various RFID Anywhere development models.

Contents

ALE applications

ALE-driven applications use the contents of ALE event cycle reports to apply business logic.

This lesson takes you through source code samples from the ALEDemo application that are useful for developing your own ALE-driven applications.

☞ For information on the ALE Application that is part of the Inventory Tracker demo, see "The ALE demo" [*RFID Anywhere Getting Started Guide*, page 205].

Note
The sample source code discussed in this lesson can be found in ALEDemo application. Open the C# project file *AleDemo.csproj*, which is located in the *Inventory Tracker\ALE Application* folder of your RFID Anywhere installation directory.

Receiving ALE reports

ALEDemo replicates an external enterprise application that is capable of receiving ALE reports from RFID Anywhere. It applies business logic to map raw tag ids to corresponding manufacturer and product names. ALEDemo is developed using C#. It is important to know that external applications which consume reports can be developed in any programming language. With RFID Anywhere, developers are not limited to a single programming environment.

In order for the ALEDemo application to be able to receive the ALE reports, it needs to open a TCP port and continuously listen on it. Remember that the ALE reports generated from the three ALE Event Cycle connectors are transmitted to external applications using the InvTracAleTCP connector.

The following code takes care of this:

Note
All code samples provided in this lesson are written in C#. As mentioned earlier, external applications can be written in any programming language.

```
private void StartTCP()
{
 try
 {
  listener = new TcpListener(IPAddress.Any, port);
  listener.Start();
  listenThread = new Thread(new ThreadStart(TcpThread));
  listenThread.Start();
 }
 catch( Exception err)
 {
  MessageBox.Show(this, err.Message, "TCP/IP",
         MessageBoxButtons.OK, MessageBoxIcon.Error);
 }
}
```

The above code opens a TCP connection that listens on the port specified by the **port** variable. The value of port must be the same as the value specified for the Port property of the TCP messaging connector.

The following methods show how to receive ALE reports and process them.

```
public void TcpThread()
{
 // Supporting variables
 byte[] bytes = new Byte[8192];
 int I = 0;
 StringBuilder XML;
 while(true)
 {
  TcpClient client = listener.AcceptTcpClient();
  // Get a stream object for reading and writing
  NetworkStream stream = client.GetStream();

  XML = new StringBuilder();
  // Read all data in the stream
  while((I = stream.Read(bytes, 0, bytes.Length)) != 0)
  {
   // Translate data bytes to a string
   XML.Append(Encoding.UTF8.GetString(bytes, 0, I));
  }
  // Send the XML documents off for processing
  ProcessXML(XML.ToString());
 }
}
```

```
protected void ProcessXML(string XML)
{
 // Check if the XML stream contains more than one XML document
 while(XML.IndexOf("<?xml") != XML.LastIndexOf("<?xml"))
 {
  // Chop off the first XML document from the string
  ProcessXMLDocument(XML.Substring(0, XML.IndexOf("<?xml", 5)));
  // Gather the remaining -1 documents into a new string
  XML = XML.Substring(XML.IndexOf("<?xml", 5));
 }
 // Deal with the final document
 ProcessXMLDocument(XML);
}
```

The first method receives ALE reports from the port and stores the information as a string. The second method takes the string and looks for multiple reports. If there are multiple reports, then it segregates them into individual reports.

Obtaining raw tag IDs from ALE reports

You were able to see the format of the ALE reports in "Using ALE to Filter and View RFID Data" [*RFID Anywhere Getting Started Guide*, page 153]. ALE reports are XML-based, and in every ALE report, the tag IDs are located in the reports ➤ report ➤ group ➤ groupList ➤ member ➤ tag element.

There are two ways that information can be retrieved from ALE reports. The first method is to treat ALE reports as generic XML documents and retrieve information by traversing each node. The second method is to deserialize and work with ALE reports as objects.

To receive ALE reports as objects, you need to set the Send Reports as Object property of the ALE Event Cycle connector to True from the Administrator Console. The following code provides an example of deserializing and working with ALE reports as objects to retrieve the raw tag IDs and the total number of groups. In each report that is received, the code iterates through all report definitions and in each report definition, it iterates through all of the groups. In each group, the code iterates through all members contained in groupList and retrieves the tag values.

☞ For a code sample of working with a generic XML document, see "Report Engine MP applications" on page 101.

```
protected void ProcessXMLDocument(string XML)
{
 long groups;

 // new instance of ECReports
 ECReports reports = ECReportsSerializer.Deserialize(XML);

 if (reports != null)
 {
  // Loop through all report definitions (report definitions are
         specified via the Administrator Console for each ALE
         Event Cycle connector)
  foreach(ECReport report in reports.reports)
  {
   groups = 0;
   // Check if there is at least one group within the report
   if( report.group != null )
   {
    // Loop through all of the groups within the specific report
    // report.group is an array containing all of the groups
         within report
    // Remember that groups are created using the Grouping
         Patterns property for each ALE Event Cycle connector.
         If no Grouping Patterns are specified, then as a
         default, one group will be created.
    foreach(ECReportGroup group in report.group)
    {
     // Obtain the group count
     int numGroups = group.groupCount.count;

        if(group.groupList.member != null)
     {
      // Loop through all of the members in the groupList. There
         will be one member for each tag id
      foreach(ECReportGroupListMember member in
         group.groupList.member)
      {
       // Check report name
       if(report.reportName == "Source1Additions")
       {
        // Get that raw tag id
        tagID = member.tag.Value;
       }
      }
     }
     groups++;
    }
   }
  }
 }
}
```

The code above gives an example of how to retrieve the raw tag IDs and the

group counts contained in ALE reports. Once the tag IDs are retrieved, they can then be compared to a secondary source file or database to get the product name or value represented by them. An example of applying this type of business logic can be found in the ALEDemo application source code.

Report Engine MP applications

Report Engine MP is used to generate reports on processed RFID tag data. Like ALE, enterprise applications need to receive the reports and apply business logic. An example of receiving reports transmitted via a TCP messaging connector is provided in "Using TCP/IP to receive reports" [*RFID Anywhere Getting Started Guide*, page 171].

Report Engine MP generates XML-based reports on RFID tags of different protocols. One of the features of Report Engine MP is that it identifies the various attributes of raw tag ids. For example, if the tags are in the SGTIN-96 EPC format, Report Engine MP reports will contain the Filter, Company Prefix, Product Name, and the Serial Number of each tag. Developers can specify a variety of encoding formats (EPC or ISO), use their own DPP, and specify complex SQL statements to filter data.

☞ To learn how to generate Report Engine MP reports and to see an example of a Report Engine MP report, see "Using Report Engine MP" [*RFID Anywhere Getting Started Guide*, page 183].

Since Report Engine MP reports are XML-based, retrieving valuable data from the reports is much like retrieving data from any generic XML document. Given the tremendous flexibility offered by Report Engine MP, it is not possible to work with these reports as objects. The following section provides an example of retrieving data from Report Engine MP reports.

```
// Declare an XML reader and give it a MemoryStream tied to the
        XML string
XmlReader reader = new XmlTextReader(new StreamReader(new
        MemoryStream(Encoding.Unicode.GetBytes(XML))));
// Scan through the XML stream

string groupName = "";
string tagCount = "";

while(reader.Read())
{
 // Get the group name.
 if (reader.LocalName == "group")
 {
  groupName = reader.GetAttribute(0);
 }
 // Get the number of tags in the group
 if (reader.LocalName == "groupCount")
 {
  tagCount = reader.ReadString();
 }
```

```
 // For each member (tag ID), read the required information
        (attributes).
 // As an example, for EPC SGTIN-96 tags, the tag information
        includes the hexadecimal encoding, EPC URI, Filter,
        CompanyPrefix, ItemReference and Serial Number
 while (reader.LocalName == "member")
 {
  lstTags.Items.Add(reader.GetAttribute(0));
  lstRawTags.Items.Add(reader.GetAttribute(1));
  lstFilter.Items.Add(reader.GetAttribute(2));
  lstCompany.Items.Add(reader.GetAttribute(3));
  lstAssetRef.Items.Add(reader.GetAttribute(4));
  reader.Read();
 }
}
```

This code gives an example of how to retrieve raw tag IDs, group counts, and other tag attributes contained in Report Engine MP reports. Once this information is retrieved, it can then be compared to a secondary source file or database to get the product name and/or apply additional business logic.

Additional business module examples

The chapter "Using the RFID Anywhere Visual Studio Extension to Create Custom Business Modules" on page 7 covers the essential components that are necessary to build a business module. This section covers topics that you might find useful when developing your own business modules. The topics discussed in this lesson include:

- Supplying help for business modules in the Administrator Console
- Preventing a business module from starting automatically
- Accessing the RFID Anywhere registry
- Communicating between business modules
- Exposing business module methods via web services
- Obtaining complete EPC tag URI from the RFID Anywhere default DPP
- Using role-based security in business modules
- Enumerating services

Each topic is discussed using sample source code where applicable.

Supplying help for business modules in the Administrator Console

When you create custom business modules for your RFID Anywhere network, you can add help for your business modules that provides information about the properties that can be configured in the Administrator Console. There are two steps required to supply help for your custom business modules:

- adding a Help property to your code
- adding the help content

Adding a Help property

If you want your custom business module to have help that is accessible from the Properties Manager, you must add a Help property to your code. The Help provided for the connectors and business modules included with RFID Anywhere is provided in HTML pages in the *docs* directory of the RFID Anywhere installation folder. The help files included with RFID Anywhere use the following naming convention:*namespace*.*Service-Type-Name*.**html**.

The RFID Anywhere documentation set includes a template HTML page (the default location is *C:\Program Files\Sybase\RFID Anywhere\docs*) that

you can use when adding custom business module help. However, the Help property can point to any HTTP URL address or a *.chm* file, so long as the file name and location specified in your code match that of the HTTP URL address or *.chm* file.

☞ For instructions on adding the HTML page with custom help content, see "Customizing the help content" on page 104.

The Help property returns a string with the name of the HTML page that contains the help. For example, the following region of code for a business module provides a link to an HTML page name named *BMnamespace.BMclassname.html* placed in the *docs* directory, where *BMnamespace* is the namespace of the business module, and *BMclassname* is the business module's class name:

```
#region Help Link
// Link to help files
[HyperLinkProperty("Help")]
public String Help
{
 get
 {
  return "docs/" + this.GetType().FullName + ".html";
 }
}
#endregion
```

Once you build the project and deploy it to your own computer or to a managed system, when a user edits the business module in the Properties Manager, a blue question mark appears beside the business module name. Users can click the blue question mark to view the help for the business module.

Customizing the help content

Help files that are made available from the Properties Manager are located in the *docs* directory of an RFID Anywhere install. In this directory, there is a file named *template.html* that you can use as a template for writing help for your custom business modules.

The template includes areas for you to provide the name of your business module, and a list of properties that can be edited in the Properties Manager, as well a description of each property.

It is recommended that you place the help file for your custom business module in the *docs* directory.

Note
The *template.html* file is installed as part of the RFID Anywhere documentation. If you do not have the RFID Anywhere documentation installed on your computer, you will not have this file.

❖ **To add help for a custom business module**

1. Open the *template.html* file in an editor. This file is located in *C:\Program Files\Sybase\RFID Anywhere\docs*.
2. Save the file with the name you specified for the Help property in your code. Make sure you save the file in the location referenced in your code.

   ☞ For more information, see "Adding a Help property" on page 103.
3. Replace the following text with information appropriate for your custom business module:
   - <**Add component name here**> Replace this text with the name of your custom component.
   - <**Insert property name**> Replace this text with the name of a property that users can edit in the Administrator Console.
   - <**Insert property description**> Replace this text with a description of the property. You can provide such information as valid values and whether or not a value is required for this property.

☞ For information about using the Properties Manager help, see "Using the Properties Manager help" [*RFID Anywhere Getting Started Guide*, page 38].

Preventing a business module from starting automatically

Business modules by default are automatically started every time the RFID Anywhere service is started. In some instances, you might not want a business module to start automatically. For example, you may want to manually start a business module at a particular time of the day or you may only want it to start after an external application is running.

Manually changing the Start Mode property

The procedure below prevents a deployed business module from automatically starting when the RFID Anywhere service is started. The business module can then be started at a later time by selecting the business module in the Administrator Console Home page and clicking Start.

❖ **To prevent business modules from starting automatically after the RFID Anywhere service is started**

1. Open the Administrator Console and log in.
2. On the Home page, select the business module from the Local Services list and then click Edit.

   The Properties Manager of the business module opens in the right pane.
3. Click the pencil beside the Start Mode property and choose Manual from the dropdown list. Click the green checkmark to save the data.

Overriding the default Start Mode property programmatically

All business modules expose three configurable OIM properties to the Administrator Console Properties Manager: Start Mode, Trace Level, and Error Notification. These properties are common to every business module and can be configured from the Administrator Console. The one property that is often useful to hardcode into the business module is Start Mode.

The Start Mode property determines whether or not the business module automatically starts when the RFID Anywhere service is started. When a business module is deployed for the *first* time, the Start Mode property is set to Automatic by default. If you do not want the business module to start automatically the *first* time it is deployed, then you can hardcode it into the business module with a single statement. Enter the following statement to your business module constructor:

```
base.StartMode = iAnywhere.RfidNet.Core.ServiceStartMode.Manual;
```

This statement sets the Start Mode property of the business module to Manual. Since it is placed in the constructor, the Start Mode property is set immediately after the business module is loaded and the Start method is not called.

Similarly, the statement below could be used to hard-code the business module's Trace Level property to Verbose.

```
base.TraceLevel =
         iAnywhere.RfidNet.Core.RnTrace.TraceLevel.Verbose;
```

Note
When a property is hard-coded, it can still be changed at a later time using the Administrator Console. However, the changes will be overwritten when the RFID Anywhere service is restarted. They will be replaced with the hard-coded values.

Limiting connectors

Limiting connectors is useful when you have a large number of RFID readers deployed on your RFID network. Your business module may only want to focus on a specific set of readers such as those located near the shipping and receiving doors. In this case, it does not make sense to connect to all the RFID readers on your network and congest your enterprise network with unnecessary data. Here, you only need the business module to connect to the required readers.

All RFID reader connectors on your network are controlled by the Multiprotocol RFID reader controller. To limit connectors in your business module, you need to attach to the Multiprotocol RFID reader controller and then connect to the required reader connectors. The following code shows you how to limit your business module to connect to just the required RFID readers.

```
private void StartControllers()
{
 try
 {
  // Attach to the Multiprotocol RFID reader controller
  mRfidMPController =
         (iAnywhere.RfidNet.Rfid.Multiprotocol.RfidMPController)
   ServiceFactory.GetService( typeof
   ( iAnywhere.RfidNet.Rfid.Multiprotocol.RfidMPController) );
  // Register to Events
  if( mRfidMPController != null )
  {
   // string array to hold readers (sources) that you want to
         connect to
   string[] Readers = {"ShippingReader1\\Ant1",
         "ReceivingReader2\\Ant4"};
   // connect to only the readers that are required
   ConnectToController( mRfidMPController, this, Readers );
   // issue read triggers on only those readers
   mRfidMPController.IssueReadTrigger( Readers )
  }
 catch( Exception ex )
 {
  throw new RnServiceStartException( "Failed to Start (" +
         ex.Message + ")" );
 }
}
```

In this code, the string `Readers` is used to specify the RFID readers that you want to receive data from. It is passed as a parameter when calling ConnectToController and IssueReadTrigger. Note that if `Readers` is replaced with `null`, then the business module connects and issues read

triggers to all RFID readers that are controlled by the Multiprotocol RFID reader controller.

Accessing the Windows registry

It is sometimes necessary to access the Windows registry for RFID Anywhere- related keys. For example, in your business module you may need to know the RFID Anywhere installation directory. The following code sample shows how to accomplish this.

```
// String to save the location of the RFID Anywhere installation
         directory
string installationDir = null;

// Access registry and retrieve the location of the RFID
         Anywhere installation directory
private void GetLocation()
{
 RegistryKey regKey = Registry.LocalMachine;
 regKey = regKey.OpenSubKey("SOFTWARE\\Sybase\\RfidAnywhere");
 installationDir =
         regKey.GetValue("Installinstallation").ToString();
}
```

Note
For the above code to compile, you need to add the **Microsoft.Win32** reference to your business module.

An example of accessing the Windows registry using the above method can be found in the source code for the Inventory Tracker business module.

Communicating between business modules

Communicating between business modules becomes essential when you have multiple business modules deployed on your RFID Anywhere network. One business module might have a method that you want to invoke from another business module. Rather than re-writing the code, RFID Anywhere allows you to access methods in other business modules easily using .NET Remoting. Much like attaching a controller to a business module, you can attach other business modules and call their methods in real-time. Remoting between business modules is trusted and as such, it does not require any security parameters.

Note
This method of communication between business modules is possible only when the business modules are deployed on the same RFID Anywhere node or when they are deployed across a non-firewalled network that permits TCP traffic.

As an example, take the GetCurrentInventory method of the Inventory Tracker business module. The GetCurrentInventory method gets a list of tags that are visible by all of the readers attached to the business module. The method returns a string containing the serial number, product name (SKU), company name and reader name for all the tags that are in the vicinity of the readers.

If a second business module requires the string returned by the GetCurrentInventory method of Inventory Tracker, the code in the following sections could be used to invoke the GetCurrentInventory method of the Inventory Tracker business module from the second business module using .NET remoting.

Communicating between business modules deployed on the same RFID Anywhere node

Communicating between business modules deployed on the same RFID Anywhere node is extremely simple. Although .NET Remoting is used under the covers, RFID Anywhere hides much of the complexity from the developer. The following instructions show how to use .NET Remoting when both business modules are deployed on the same RFID Anywhere node.

❖ To call the GetCurrentInventory method from your new business module

1. From Visual Studio .NET, open the new business module project.
2. From the Project menu, choose Add Reference.

   The Add Reference window opens.
3. Click Browse and select *InventoryTracker.dll.*
4. Click Open and then click OK to close the Add Reference window.
5. Add the following `using` directive at the beginning of the source code:

   ```
   using inventoryTracker;
   ```

   This adds the inventoryTracker namespace for use with your business module.

6. You need to modify your business module declaration to add a dependency to your business module. To do this, set the last parameter of the business module declaration to the GUID of the Inventory Tracker business module. For example, if your business module is an RnWebBusinessModule (exported as a web service):

```
[RnWebBusinessModule("BA6C54E7-8CA2-3214-8CE2-B3E2A1123A5B",
         "BM that invokes Inventory
 Tracker BM" , new string[]{"AA6B44F6-9DB6-4961-9DC1-
         D4D2F6788D0C"})]
```

   If your business modules is a RnBusinessModule:

```
[RnBusinessModule("BA6C54E7-8CA2-3214-8CE2-B3E2A1123A5B",
         "BM that invokes Inventory
 Tracker BM", false, new string[]{"AA6B44F6-9DB6-4961-9DC1-
         D4D2F6788D0C"})]
```

   This code specifies that the new business module is dependant on the Inventory Tracker business module, meaning that the Inventory Tracker business module will start before your business module. This allows your new business module to invoke the GetCurrentInventory method of Inventory Tracker even when Inventory Tracker is not started from the Administrator Console. If this modification is not performed (the value for dependency is null), then you need to make sure that the Inventory Tracker business module is started before you invoke one of its methods from the new business module.

7. Declare an InventoryTracker variable above your business module Start method.

```
private inventoryTracker.InventoryTracker invTrac;
```

8. Add the following code in your business module Start method, below `base.Start( context );`:

```
invTrac = (InventoryTracker)ServiceFactory.GetService(
          typeof( InventoryTracker ) );
          if( invTrac == null ) throw new RnException(
          "invTrac Service Module not found" );
```

   The above code attaches the Inventory Tracker business module to your new business module. If Inventory Tracker is not found, it catches an error.

9. Call the GetCurrentInventory method of InventoryTracker.

```
string currInv = invTrac.GetCurrentInventory();
```

   Since you know that GetCurrentInventory method returns a string, you can store the value in the locally declared currInv string.

In this example, you accessed the GetCurrentInventory method of Inventory Tracker business module. RFID Anywhere allows you to access any `public` method of a business module. For example, accessing the Start method from your business module will allow you to modify the state of an attached business module.

Communicating with ALE from a business module

To connect to the ALEService Web Service from your custom business module, you perform similar steps to those found in the section above. ALEService is a business module that exposes ALE methods via Web services.

The section below outlines how you would attach to the ALEService Web Service from your business module.

```
private iAnywhere.RfidNet.Ale.ALEService mAleServ;

mAleServ = (ALEService)ServiceFactory.GetService( typeof(
         ALEService ) );
```

You can also connect directly to the ALE controller from your business module by selecting the Application Level Event controller from the Add Controller option of the RFID Anywhere menu from the RFID Anywhere Visual Studio Extension.

Obtaining complete EPC tag URI from the default DPP

The RFID Anywhere default DPP provides a method to obtain the complete EPC tag URI. “Creating and Using a Data Protocol Processor (DPP)” on page 65 introduced you to Data Protocol Processor (DPP) and how to use the RFID Anywhere default DPP to encode and decode tags. It showed you how to access the various attributes of an EPC tag individually. But what if you want to get the entire EPC tag URI in a single step, as opposed to getting the attributes separately and then concatenating them? This section shows how you can obtain a complete tag URI in a single step using the RA default DPP.

Once an encoded hexadecimal tag has been stored and decoded in an RnInfo object, you can obtain the complete tag URI by using the RfidEPCTagInfo class.

Declare an RfidEPCTagInfo object.

```
RfidEPCTagInfo epcInfo = new RfidEPCTagInfo();
```

Suppose that the decoded EPC tag is stored in an RnInfo object called info. Set epcInfo to info by casting it.

```
epcInfo = (RfidEPCTagInfo)info;
```

Obtain the complete EPC tag URI and store it to a string.

```
string URI = epcInfo.GetProperty("Uri").ToString();
```

☞ For more information on DPP, RFID Anywhere default DPP, and how to use the default DPP to store and decode EPC tags, see “Creating and Using a Data Protocol Processor (DPP)” on page 65.

Using role-based security in business modules

This section shows you how to verify user roles within a web method. This type of security applies for web method calls only and does not apply when a business module method is invoked via .NET remoting. Remoting between business modules is fully trusted and does not include any security.

Suppose you have a business module that exposes several methods via web services and that certain methods can only be accessed by Administrators. When external applications invoke one of the web methods, you need to check the role of the user and allow access accordingly. The following sample code provides an example of verifying user roles. The code must be entered at the beginning of your webmethod.

```
if (System.Threading.Thread.CurrentPrincipal is
        iAnywhere.RfidNet.Security.ADSI.SecurityPrincipal)
{
 iAnywhere.RfidNet.Security.ADSI.SecurityPrincipal principal =
  (iAnywhere.RfidNet.Security.ADSI.SecurityPrincipal)
  System.Threading.Thread.CurrentPrincipal;
 // Check whether the user is in the Administrator group
 bool blsInRole =
        principal.IsInRole(System.Security.Principal.WindowsBui
        ltInRole.Administrator);
 // If the user is in Administrator group, then allow access to
        the webmethod
 if (blsInRole)
 {
  // The rest of your webmethod
 }
}
```

Enumerating services

It is often useful to obtain information about the controllers and business modules installed on an RFID Anywhere node. For example, if your business module depends on another business module, then you may want to verify whether the other business module is installed on the node. Instead of manually checking using the Administrator Console, you can implement this functionality in the dependent business module.

The sample code below shows you how to enumerate through all of the services installed on an RFID Anywhere node and retrieve specific attributes.

```
public override void Start(RnContext context)
{
 try
 {
  base.Start( context );

  // Init Controllers
  StartControllers();
    if(context!=null)
  {
   System.Collections.ArrayList al = context.services;
   if(al!=null)
   {
    foreach(RnServiceInfo a in al)
    {
     TraceLog.WriteInfo("Assembly: "+a.AssemblyName);
     TraceLog.WriteInfo("Description: "+a.Description);
     TraceLog.WriteInfo("Description: "+a.Id);
     TraceLog.WriteInfo("Description: "+a.Location);
     TraceLog.WriteInfo("Description: "+a.TypeName);
     TraceLog.WriteInfo("Description: "+a.Url);
    }
   }
   else
    TraceLog.WriteError("No services");
  }
  else
   TraceLog.WriteError("Null context");
 }
 catch( Exception ex )
 {
  throw new RnServiceStartException( "Failed to Start (" +
         ex.Message + ")" );
 }
}
```

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X