RFID SDK Windows Developer Guide

Contents

RFID SDK Windows Developer Guide

RFID SDK Windows Developer Guide

The *RFID SDK Windows Developer Guide* provides installation and programming information that allows RFID application development for Windows 7/10/11.

Related Documents

- RFID Scanner SDK for Windows API Reference Guide
- RFD8500 User Guide, p/n MN002065Axx
- RFD8500i User Guide, p/n MN-002761-XX.
- RFD8500 Quick Start Guide, p/n MN002225Axx.
- RFD8500i Quick Start Guide, p/n MN-002760-XX
- RFD8500 Regulatory Guide, p/n MN002062Axx.
- RFD8500i Regulatory Guide, p/n MN-002856-xx.
- RFD8500/iRFID DEVELOPER GUIDE

Overview

This provides step-by-step instructions to import the RFID SDK module and build Windows applications (with Microsoft .NET 4.8) to work with RFID readers.

To build Microsoft .NET 4.8 applications for Windows use Visual Studio 2017 or above.

Creating a Windows Project

To create a C# Windows Project in Visual Studio 2017:

1. Start Visual Studio 2017->Select File > New > Project->Visual C#, create a new Windows Forms Application project and follow the on-screen steps in Visual Studio.

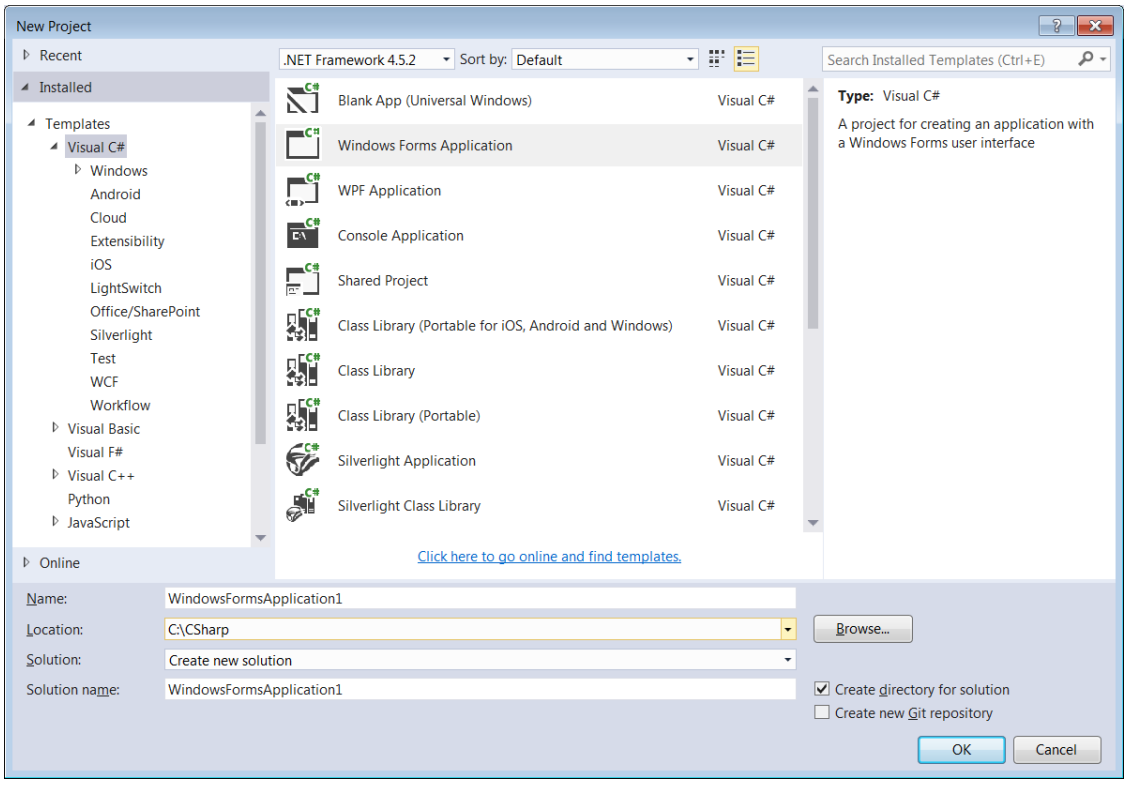

Figure 1-1 *Create New Project*

2. Add a reference to the **Symbol.RFID.SDK** and **Symbol.RFID.SDK.Domain.Reader** assemblies/DLLs from RFID SDK binaries.

3. Import the **Symbol.RFID.SDK** and **Symbol.RFID.SDK.Domain.Reader** namespace/classes.

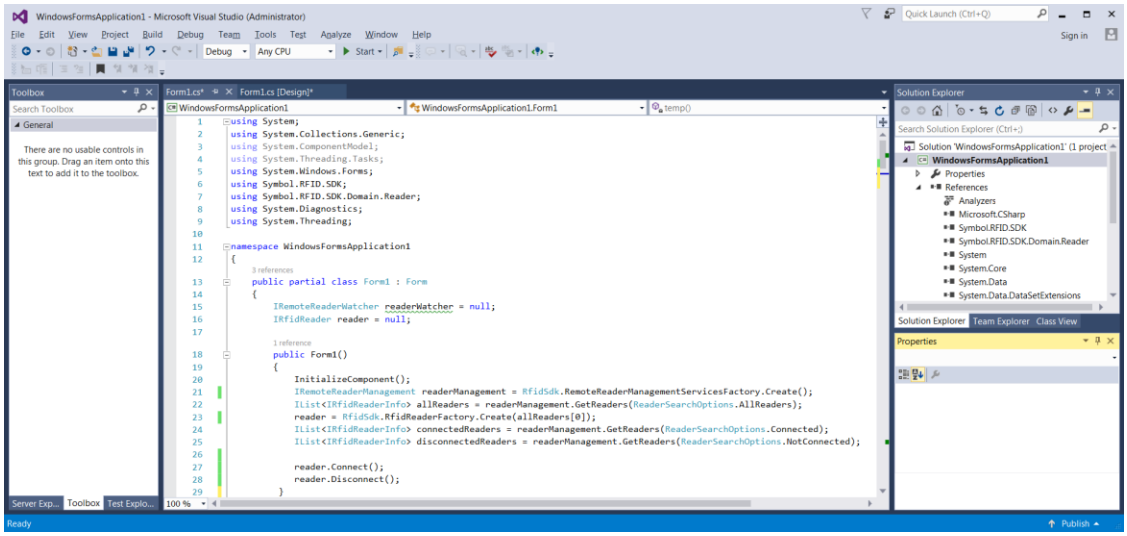

Figure 1-2 *Example Application with RFID SDK*

Building and Running a Project

Before building/running make sure the following assemblies are in the target application folder. This is the folder where the compiled application resides (Ex \bin\Debug\) folder:

- InTheHand.Net.Personal.dll
- RFIDCommandLib.dll
- Symbol.Extensions.Compatibility.dll
- Symbol.RFID.SDK.Connectivity.Windows.dll
- Symbol.RFID.SDK.Discovery.Windows.dll
- Symbol.RFID.SDK.dll
- Symbol.RFID.SDK.Domain.Reader.dll
- Symbol.RFID.SDK.Domain.Reader.Infrastructure.dll
- Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.dll
- Symbol.RFID.SDK.IP.dll
- Symbol.RFID.SDK.Logger.dll
- Symbol.RFID.SDK.USB.dll

Add the following settings to the App.Config file (ensure the correct paths are specified for the target assemblies).

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8"/>
  </startup>
  <appSettings>

    <add key="RemoteReaderAssembly" value="Symbol.RFID.SDK.Domain.Reader.dll"/>
    <add key="RemoteReaderService" value="Symbol.RFID.SDK.Domain.Reader.ZetiRfidReader"/>
    <add key="RemoteReaderInfrastructureServiceAssembly"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.dll"/>
    <add key="RemoteReaderInfrastructureService"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.ZetiRfidReaderAdapter"/>

    <!--Bluetooth Device Settings-->
    <add key="RemoteReaderConnectionAssembly" value="Symbol.RFID.SDK.Connectivity.Windows.dll"/>
    <add key="RemoteReaderConnectionService"
         value="Symbol.RFID.SDK.Connectivity.Windows.SocketDeviceConnection"/>

    <add key="RemoteReaderDiscoveryServiceAssembly"
         value="Symbol.RFID.SDK.Discovery.Windows.dll"/>
    <add key="RemoteReaderDiscoveryService"
         value="Symbol.RFID.SDK.Discovery.Windows.ReaderWatcher"/>

    <add key="RemoteReaderManagementServiceAssembly"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.dll"/>
    <add key="RemoteReaderManagementService"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.RemoteReaderManagement"/>
```

```
    <!--USB Device Settings-->
    <add key="UsbReaderConnectionAssembly" value="Symbol.RFID.SDK.Connectivity.Windows.dll"/>
    <add key="UsbReaderConnectionService"
         value="Symbol.RFID.SDK.Connectivity.Windows.UsbSerialPortDeviceConnection"/>

    <add key="UsbReaderDiscoveryServiceAssembly" value="Symbol.RFID.SDK.Discovery.Windows.dll"/>
    <add key="UsbReaderDiscoveryService"
         value="Symbol.RFID.SDK.Discovery.Windows.UsbReaderWatcher"/>

    <add key="UsbReaderManagementServiceAssembly"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.dll"/>
    <add key="UsbReaderManagementService"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.UsbReaderManagement"/>

    <!--IP Device Settings-->
    <add key="IpReaderConnectionAssembly" value="Symbol.RFID.SDK.Connectivity.Windows.dll"/>
    <add key="IpReaderConnectionService"
         value="Symbol.RFID.SDK.Connectivity.Windows.IPSocketDeviceConnection"/>

    <add key="IpReaderDiscoveryServiceAssembly" value="Symbol.RFID.SDK.Discovery.Windows.dll"/>
    <add key="IpReaderDiscoveryService"
         value="Symbol.RFID.SDK.Discovery.Windows.IPReaderWatcher"/>

    <add key="IpReaderManagementServiceAssembly"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.dll"/>
    <add key="IpReaderManagementService"
         value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management.IPReaderManagement"/>

    <!--When the following key, the value pair not present in the configuration logging will be
        disabled. To enable logging set the value to "true" and otherwise "false".-->
    <add key="LogEnabled" value="true"/>
    <add key="LogNamespace" value="cnn"/>
  </appSettings>
</configuration>
```

NOTE: Make sure the specified DLLs are present in the target application folder, add the above `<appSettings>` section is in the App.Config xml configuration file of the project.

Enabling Logging

When logging is enabled and the above key-value pair is not present, it will log all possible logs. To enable logging for specified libraries, put the comma separated (cnn, dsc) library key values as in the following list:

Library	Key
All logs	- all
Symbol.RFID.SDK	- sdk
Symbol.RFID.SDK.Connectivity.Windows	- cnn
Symbol.RFID.SDK.Discovery.Windows	- dsc
Symbol.RFID.SDK.Domain.Reader	- rdr
Symbol.RFID.SDK.Domain.Reader.Infrastructure	- inf
Symbol.RFID.SDK.Domain.Reader.Infrastructure.Management	- mng
Symbol.RFID.SDK.USB	- usb

Pairing with Bluetooth

Pairing with a Personal Computer

1. If the BT LED is not blinking, press the BT button for 1 second to make the RFD8500 discoverable (the BT LED starts blinking when in discoverable mode). From the *Start* menu, select *Device and Printers.* Select **Add a device**.

2. Select the device and click **Next**. When the BT LED starts blinking rapidly press the trigger within 25 seconds to acknowledge pairing.

3. Click **Close** to complete the pairing process.

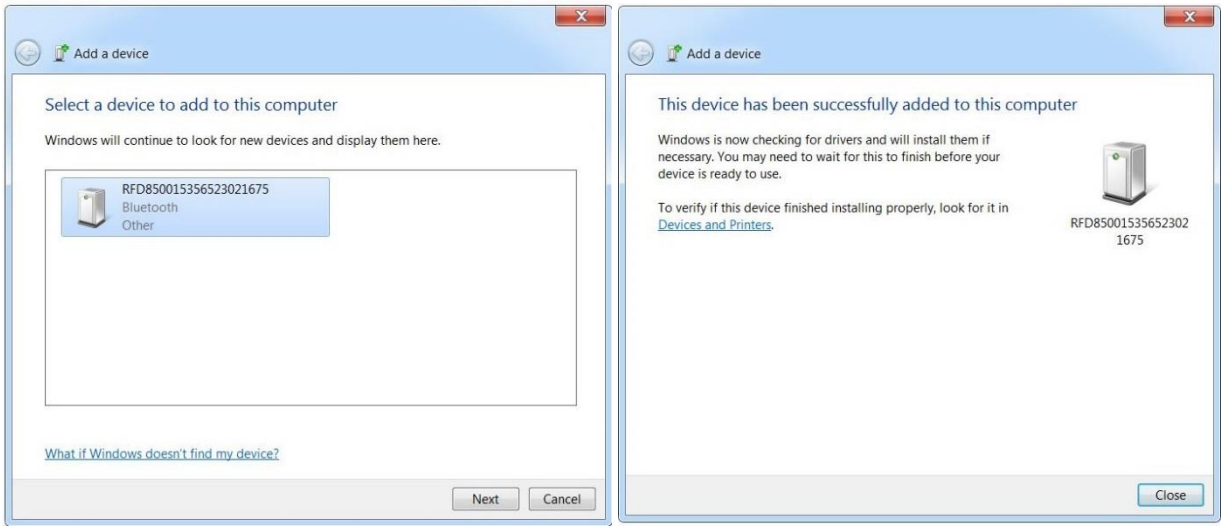

Figure 1-3 *Bluetooth Pairing with Host*

Zebra RFID SDK for Windows Overview

This chapter provides detailed information about how to use various functionalities (basic to advanced) of the Windows RFID SDK to develop applications.

The Zebra RFID SDK for Windows provides an API that can be used by external applications to manage and control RFID specific functionality of a connected RFID reader over Bluetooth/USB-CDC/IP.

The supported RFID readers and corresponding communication modes are as follows:

- RFD8500: Bluetooth
- RFD40-Standard: USB-CDC
- RFD40 Premium: Bluetooth,USB-CDC
- RFD40 Premium Plus: Bluetooth,USB-CDC,IP
- RFD90: Bluetooth,USB-CDC, IP

The Zebra RFID SDK for Windows provides the ability to manage RFID readers' connections, perform various operations with connected RFID readers, configure connected RFID readers and retrieve other information related to connected RFID readers.

All available APIs are defined under the **Symbol.RFID.SDK** namespace. The application uses the interface **IRfidReader** to interact with a reader.

Use available **IRfidReader** interface to register for events, connect with readers, and after successful connection perform required operations such as inventory.

If method calls fail, the corresponding method throws an exception. The application should call all API methods in try-catch blocks for handling exceptions.

Connect to a RFID Reader

Connection is the first step to communicate with a RFID reader. Import the namespace to use the RFID API as shown below.

```
using Symbol.RFID.SDK;
using Symbol.RFID.SDK.Domain.Reader;
```

Create an **IRemoteReaderManagement** interface instance by using the **RfidSdk.ReaderManagementServicesFactory** class Create method (with the communication mode) as follows:

```
IRemoteReaderManagement readerManagement =
          RfidSdk.ReaderManagementServicesFactory.Create(ReaderCommunicationMode.Bluetooth);
```

Next call **GetReaders** method of the **IRemoteReaderManagement** interface instance object that gives a list of all available/paired RFID readers with a Windows device/PC. Readers list is in the form of **IRfidReaderInfo** interface instance collection.

```
IList<IRfidReaderInfo>allReaders =
          readerManagement.GetReaders(ReaderSearchOptions.AllReaders);
```

The following table lists the reader search options that can be specified as a parameter to **GetReaders** method.

Reader Search Options	Description
`ReaderSearchOptions.AllReaders`	Gives a list of all available/paired RFID readers with a Windows device/PC
`ReaderSearchOptions.Connected`	Gives a list of all connected RFID readers with a Windows device/PC
`ReaderSearchOptions.NotConnected`	Gives a list of all paired but not connected RFID readers with a Windows device/PC

Next call the **RfidSdk.RFIDReaderFactory.Create** method with the **IRfidReaderInfo** instance of the device to communicate with the device as follows.

```
IRfidReader reader = RfidSdk.RfidReaderFactory.Create(allReaders[0]);
```

The returned **IRfidReader** reader interface is used for performing all operations with RFID reader. To connect with the reader; use **IRfidReader** instance **Connect()** method.

```
// Establish connection to the RFID Reader
reader.Connect();
```

Initializing RFID Reader Watchers

RFID Reader Watchers for Bluetooth/USB-CDC/IP can be initialized as shown below. The application can register for **IRemoteReaderWatcher** instance events in the following way to get notified of RFID readers getting added (paired) / removed(unpaired), connected/disconnected.

```
/// <summary>
/// Initialize the reader watcher events.
/// </summary>
private void InitializeReaderWatcher()
{
    try
    {
        IRemoteReaderWatcher remoteReaderWatcher =
        RfidSdk.ReaderWatcherServicesFactory.Create(ReaderCommunicationMode.Bluetooth);
        remoteReaderWatcher.ReaderAppeared += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderAppearedHandler);
        remoteReaderWatcher.ReaderDisappeared += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderDisappearedHandler);
        remoteReaderWatcher.ReaderConnected += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderConnectedHandler);
        remoteReaderWatcher.ReaderDisconnected += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderDisconnectedHandler);
        isBluetoothWatcherInitialized = true;
    }
    catch (Exception ex)
    {
        // Print exception to log if Initializing ReaderWatcher failed
        Debug.WriteLine(ex.Message);
    }


    try
    {
        IRemoteReaderWatcher usbReaderWatcher =
        RfidSdk.ReaderWatcherServicesFactory.Create(ReaderCommunicationMode.USB);
        usbReaderWatcher.ReaderAppeared += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderAppearedHandler);
        usbReaderWatcher.ReaderDisappeared += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderDisappearedHandler);
        usbReaderWatcher.ReaderConnected += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderConnectedHandler);
        usbReaderWatcher.ReaderDisconnected += new
        EventHandler<ReaderStatusChangedEventArgs>(ReaderDisconnectedHandler);
        isUsbWatcherInitialized = true;
    }
    catch (Exception ex)
    {
        // Print exception to log if Initializing ReaderWatcher failed
        Debug.WriteLine(ex.Message);
    }


    try
    {
        IRemoteReaderWatcher ipReaderWatcher =
        RfidSdk.ReaderWatcherServicesFactory.Create(ReaderCommunicationMode.IP);
```

```
            ipReaderWatcher.ReaderAppeared += new
            EventHandler<ReaderStatusChangedEventArgs>(ReaderAppearedHandler);
            ipReaderWatcher.ReaderDisappeared += new
            EventHandler<ReaderStatusChangedEventArgs>(ReaderDisappearedHandler);
            ipReaderWatcher.ReaderConnected += new
            EventHandler<ReaderStatusChangedEventArgs>(ReaderConnectedHandler);
            ipReaderWatcher.ReaderDisconnected += new
            EventHandler<ReaderStatusChangedEventArgs>(ReaderDisconnectedHandler);
            isIPWatcherInitialized = true;
        }
        catch (Exception ex)
        {
            // Print exception to log if Initializing ReaderWatcher failed
            Debug.WriteLine(ex.Message);
        }
    }
```

Discover RFID Readers

Bluetooth/USB-CDC/IP RFID Readers can be discovered as follows:

```
        try
        {
            readerList = new List<IRfidReader>();

            // Initialize reader management and reader info list.
            var readerInfoList = new List<IRfidReaderInfo>();

            if (isBluetoothWatcherInitialized)
            {
               var readerManager =
               RfidSdk.ReaderManagementServicesFactory.Create(ReaderCommunicationMode.Bluetooth);
               // Get available readers list
               var remoteReaderInfoList = readerManager.GetReaders(ReaderSearchOptions.AllReaders);
               readerInfoList.AddRange(remoteReaderInfoList);
            }

            if (isUsbWatcherInitialized)
            {
               var readerManager =
               RfidSdk.ReaderManagementServicesFactory.Create(ReaderCommunicationMode.USB);
               // Get available readers list
               var remoteReaderInfoList = readerManager.GetReaders(ReaderSearchOptions.AllReaders);
               readerInfoList.AddRange(remoteReaderInfoList);

            }

            if (isIPWatcherInitialized)
            {
               var readerManager =
               RfidSdk.ReaderManagementServicesFactory.Create(ReaderCommunicationMode.IP);
               // Get available readers list
               var remoteReaderInfoList = readerManager.GetReaders(ReaderSearchOptions.AllReaders);
               readerInfoList.AddRange(remoteReaderInfoList);

             }


            foreach (IRfidReaderInfo readerInfo in readerInfoList.OrderBy(o =>
            o.FriendlyName))
            {
              // Create reader instance based on reader information.
              var reader = RfidSdk.RfidReaderFactory.Create(readerInfo);
```

```
                // Creating reader device info object
                var deviceInfo = new RFIDReaderItemInfo()
                {
                   HostAddress = readerInfo.ID,
                   PortNumber = readerInfo.ComPortNumber,
                   FriendlyName = reader.FriendlyName,
                   CommunicationMode = readerInfo.CommunicationMode,
                   Status = readerInfo.Status,
                   Reader = reader
                 };
            }

        }
        catch (Exception ex)
        {
            Debug.WriteLine("Error discovering scanner: " + ex.Message);
        }
```

Special Connection Handling Cases

In a normal scenario, the reader connects fine, but following are the cases which require special handling at the time of connection.

The following example shows a connection handled under try-catch block.

```
try
{
    // Establish connection to the RFID Reader
    reader.Connect();
}
catch (Exception e)
{
    Debug.Print(e.Message);
}
```

Region Is Not Configured

If the region is not configured an exception **ERROR_REGION_NOT_CONFIGURED** is given.

Then the caller chooses the operation regulatory region and sets the region with required configurations, as shown below:

```
private void GetRegionInfo()
{
    try
    {
        RegulatoryConfig regConfig = reader.Configurations.RegulatoryConfig;
        Debug.WriteLine("Config.RegulatoryConfig.Region : " + regConfig.Region);
    }
    catch (Exception ex)
    {
        if (ex.Message == reader.Configurations.ERROR_REGION_NOT_CONFIGURED)
            ConfigureDefaultRegion();
        else
            Debug.WriteLine(ex.Message + Environment.NewLine);
    }
}
```

```
private void ConfigureDefaultRegion()
{
    Debug.WriteLine("Region: Not Configured. Configuring as USA");
    RegulatoryConfig config = new RegulatoryConfig();
    config.Region = "USA";
    reader.Configurations.RegulatoryConfig = config;
}
```

Disconnect

When the application is done with the connection and operations on the RFID reader, call the following method to close the connection.

```
// Disconnects reader
reader.Disconnect();
```

Reader Capabilities

The capabilities (or Read-Only properties) of the reader include the following:

General Capabilities

- Model Name.
- Serial Number
- Manufacturer Name
- Manufacture Date
- Number of antennas supported.
- Is Tag Event Reporting Supported - Indicates the reader's ability to report tag visibility state changes (New Tag, Tag Invisible, or Tag Visibility Changed).
- Is Tag Locationing Supported - Indicates the reader's ability to locate a tag.
- Is Hopping Enabled

Gen2 Capabilities

- Block Erase - supported
- Block Write - supported
- State Aware Singulation - supported
- Maximum Number of Operation in Access Sequence
- Maximum Pre-filters allowable per antenna
- RF Modes.

Regulatory Capabilities

- Country Code
- Communication StandardRegion
- Hopping
- Enable Channels

For setting/getting Region see previous section: ”Region not Configured”.

Retrieving Reader Capabilities

```
// Get Reader capabilities
Console.WriteLine("ModelName: " + reader.Capabilities.ModelName);
Console.WriteLine("SerialNumber: " + reader.Capabilities.SerialNumber);
Console.WriteLine("Manufacture Name: " + reader.Capabilities.ManufactureName);
Console.WriteLine("Manufacturing Date: " + reader.Capabilities.ManufacturingDate);
Console.WriteLine("Hopping Enabled: " + reader.Capabilities.IsHoppingEnabled);
```

Configuring the reader

Antenna Specific Configuration

The **reader.Configurations** class contains the **Antennas** as object. The individual antenna can be accessed and configured using the index.

The **reader.Configurations.Antennas[antennaID].Configuration** Properties is used to set the antenna configuration to individual antenna.

The antenna configuration comprised of Transmit Power Index, Receive Sensitivity Index and Transmit Frequency Index.

Set/Get individual antenna configuration settings as follows:

```
ushort PowerVal = 270;
ushort curAntennaID = 0;
AntennaConfiguration antConfig = reader.Configurations.Antennas[curAntennaID].Configuration;
antConfig.TransmitPowerIndex = PowerVal;
reader.Configurations.Antennas[curAntennaID].Configuration = antConfig;
Console.WriteLine("Set TransmitPowerIndex = " + antConfig.TransmitPowerIndex.ToString());
```

Singulation Control

The property **SingulationControl** sets/gets the current settings of singulation control from the reader, for the given Antenna ID.

The following settings can be configured:

- Session: Session number to use for inventory operation.
- Tag Population: An estimate of the tag population in view of the RF field of the antenna.

- Tag Transit Time: An estimate of the time a tag typically remains in the RF field.

```
ushort curAntennaID = 0;
// Get Singulation
SingulationControl singulationControl=reader.Configurations.Antennas[curAntennaID].SingulationControl;
Console.WriteLine("Session : "+singulationControl.Session.ToString());
Console.WriteLine("Population : "+singulationControl.TagPopulation.ToString());
Console.WriteLine("TagTransitTime :"+singulationControl.TagTransitTime.ToString());

// Set Singulation
singulationControl.Session = SESSION.SESSION_S1;
singulationControl.TagPopulation = 30;
reader.Configurations.Antennas[curAntennaID].SingulationControl = singulationControl;

Console.WriteLine("SetSingulation :  Session  = [" + singulationControl.Session + "]");
Console.WriteLine("SetSingulation : TagPopulation:"+ singulationControl.TagPopulation.ToString());
```

Tag Report Configuration

The SDK provides an ability to configure a set of fields to be reported in a response to an operation by a specific active RFID reader.

Supported fields that might be reported include the following:

- First seen time
- Last seen time
- PC value
- RSSI value
- Phase value
- Channel index
- Tag seen count.

The **reader.Configurations.ReportConfig** class can be used to retrieve and sets the tag report parameters from the reader.

Dynamic Power Management Configuration

The SDK provides a way to configure the reader to operate in dynamic power mode. The dynamic power state can be switched to be either on or off.

```
// set Dynamic power state on
reader.Configurations.DynamicPowerConfig.setDPOState(DYNAMIC_POWER_OPTIMIZATION.ENABLE);
// set Dynamic power state off
reader.Configurations.DynamicPowerConfig.setDPOState(DYNAMIC_POWER_OPTIMIZATION.DISABLE);
```

Regulatory Configuration

The SDK supports managing of regulatory related parameters of a specific active RFID reader.

Regulatory configuration includes the following:

- Code of selected region
- Hopping
- Set of enabled channels

A set of enabled channels includes only such channels that are supported in the selected region. If hopping configuration is not allowed for the selected regions, a set of enabled channels is not specified.

Regulatory parameters could be retrieved and set via **RegulatoryConfig** property accordingly. The region information is retrieved using **Region** property. The following example demonstrates retrieving of current regulatory settings and configuring the RFID reader to operate in one of supported regions.

```
// Get Regulatory Config
RegulatoryConfig regConfig = reader.Configurations.RegulatoryConfig;
Console.WriteLine("Config.RegulatoryConfig.Region : " + regConfig.Region);
Console.WriteLine("Config.RegulatoryConfig.Hopping : " + regConfig.Hopping);
string[] enabledChannels = regConfig.EnabledChannels;

// Set Regulatory Config
RegulatoryConfig config = new RegulatoryConfig();
config.Region = "USA";
reader.Configurations.RegulatoryConfig = config;
```

Saving Configuration

Various parameters of a specific RFID reader configured via SDK are lost after the next power down. The SDK provides an ability to save a persistent configuration of RFID reader. The **SaveConfig** method can be used to make the current configuration persistent over power down and power up cycles. The following example demonstrates utilization of mentioned method.

```
// Saving the configuration
reader.Configurations.SaveConfig();
```

Reset Configuration to Factory Defaults

The SDK provides a way to reset the RFID reader to the factory default settings. The **ResetFactoryDefaults** method can be used to attain this functionality. Once this method is called, all the reader settings like events, singulation control, etc. will revert to default values and the RFID reader reboots. A connected application shall lose connectivity to the reader and must connect back again and is required to redo the basic steps for initializing the reader. The following example demonstrates utilization of this method.

```
reader.ResetFactoryDefaults(); // Resetting the configuration
```

Managing Events

The Application can register for one or more events, to be notified when it occurs. There are several types of events. The table below lists the events supported.

Event	Description
`readerWatcher.ReaderConnected`	Event notifying connection from the Reader.
`readerWatcher.ReaderDisconnected`	Event notifying disconnection from the Reader. The application can call connect method periodically to attempt reconnection or call disconnect method to cleanup and exit.
`readerWatcher.ReaderAppeared`	Event notified when reader paired.
`readerWatcher.ReaderDisappeared`	Event notified when reader unpaired.
`reader.Inventory.TagDataReceived`	Tag Data received event.
`reader.Inventory.InventoryStarted`	Inventory operation started. In case of periodic trigger, this event is triggered for each period.
`reader.Inventory.InventoryStopped`	Inventory operation has stopped. In case of periodic trigger this event is triggered for each period.
`reader.Inventory.InventorySessionSummary`	Event generated when operation end summary has been generated. The data associated with the event contains total rounds, total number of tags and total time in micro secs.
`reader.BatteryStatusNotification`	Events notifying different levels of battery, state of the battery, if charging or discharging.
`reader.PowerStatusNotification`	Events which notify the different power states of the reader device. The event data contains cause, voltage, current and power.
`reader.TemperatureStatusNotification`	When temperature reaches threshold level, this event is generated. The event data contains source name (PA/Ambient).
`reader.Inventory.BatchMode`	Event generated when batch tag read operation is in progress.
`reader.WpaConfigurationNotification`	Event generated when on WPA configuration: connect/scan notifications.
`reader.TriggerStatusNotification`	Event providing information on trigger status (trigger-type/trigger-mode).

Registering for tag data notification

```
// Registering for read tag data notification
reader.Inventory.TagDataReceived += Inventory_TagDataReceived;

private void Inventory_TagDataReceived(object sender, TagDataReceivedEventArgs e)
{
    Console.WriteLine("Events_ReadNotify --------");
    Console.WriteLine("Tag ID:" + e.EPCId);
    Console.WriteLine("Tag Seen Count:" + e.TagSeenCount);
    Console.WriteLine("RSSI" + e.RSSI);
}
```

Device Status Related Events

Device status, like battery, power, and temperature, is obtained through events after initiating the **reader.Configurations.GetDeviceStatus** method.

Response to the above method comes as battery event, power event and temperature event according to the set boolean value in the respective parameters. The following is an example of how to get these events.

```
reader.BatteryStatusNotification += Reader_BatteryStatusNotification;
reader.TemperatureStatusNotification += Reader_TemperatureStatusNotification;
reader.PowerStatusNotification += Reader_PowerStatusNotification;

bool battery = true;
bool power = true;
bool temperature = true;
reader.Configurations.GetDeviceStatus(battery, power, temperature);

private void Reader_BatteryStatusNotification(object sender, BatteryStatusNotificationReceivedEventArgs e)
{
    // Handle battery event notification
}

private void Reader_PowerStatusNotification(object sender, PowerStatusNotificationReceivedEventArgs e)
{
    // Handle power event notification
}

private void Reader_TemperatureStatusNotification(object sender,
TemperatureStatusNotificationReceivedEventArgs e)
{
    // Handle temperature event notification
}
```

Trigger Status Notification Related Events

Trigger status notification events (with information on trigger-mode/trigger-type) are obtained by registering for **reader.TriggerStatusNotification** event. The following is an example of how to receive trigger status notification events:

```
reader.TriggerStatusNotification += Reader_TriggerStatusNotification;

private void Reader_TriggerStatusNotification(object sender, TriggerStatusNotificationReceivedEventArgs e)
{
  Log($"[Notification] Trigger Evant: Trigger Mode:{e.TriggerMode}, Trigger Type:{e.TriggerType}");
}
```

Basic Operations

Tag Storage Settings

This section covers the basic/simple operations that an application would need to perform on an RFID reader, which includes inventory and single tag access operations.

Each tag has a set of associated information along with it. During the Inventory operation the reader reports the EPC-ID of the tag however during the Read-Access operation the requested Memory Bank Data is also reported apart from EPC-ID. In either case, there is additional information like PC-bits, RSSI, last time seen, tag seen count, etc. that is available for each tag. This information is reported to the application as TagData for each tag reported by the reader. Applications can also choose to enable/disable reporting certain fields in TAG_DATA. Disabling certain fields can sometimes improve the performance as the reader and the SDK are not processing that information.

Following are a few use-cases that get tags from the reader.

Reading Tag Data from Event

A simple continuous inventory operation reads all tags in the field of view of all antennas of the connected RFID reader. The start and stop trigger for the inventory is the default (i.e., start immediately when **reader.Inventory.Perform** is called, and stop immediately when **reader.Inventory.Stop** is called).

```
// registering for read tag data notification
reader.Inventory.TagDataReceived += Inventory_TagDataReceived;

// perform simple inventory reader
reader.Inventory.Perform();

// Keep getting tags in the TagDataReceived event if registered
Thread.Sleep(5000); // Wait for 5 seconds for tags to read

// stop the inventory
reader.Inventory.Stop();
private void Inventory_TagDataReceived(object sender, TagDataReceivedEventArgs e)
{
    Console.WriteLine("Events_ReadNotify --------");
    Console.WriteLine("Tag ID:" + e.EPCId);
    Console.WriteLine("Tag Seen Count:" + e.TagSeenCount);
    Console.WriteLine("RSSI" + e.RSSI);
}
```

Reading Tag Data from Queue

The **GetNextTagDataReceived()** method is used to read tag data from internal queue. This is a blocking method that retrieves oldest **ITagData** buffered in the internal queue. If no tag data is present, the method blocks and waits until tag data is received.

If a timeout is specified as a parameter the method blocks and waits for the specified amount of time, for tag data (**ITagData**) to appear in the internal queue and returns the corresponding value.

To enable tag data to be received from internal queue, update the App.Config xml <appSettings> section as follows:

```
<!--Support tag data queuing -->
<add key="ZetiResponseDispatcherAssembly"
value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.dll"/>
<add key="ZetiTagDataDispatcher"
value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.ZetiTagDataQueuingDispatcher"/>
```

NOTE: Tag data will not be returned in the form of an event when the above setting is enabled.

To enable tag data to be received as events use the following App.Config xml <appSettings> section:

```
<!--Support tag data dispatching via events for Win 7 and above or powerful WM devices -->
<add key="ZetiResponseDispatcherAssembly"
value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.dll"/>
<add key="ZetiTagDataDispatcher"
value="Symbol.RFID.SDK.Domain.Reader.Infrastructure.ZetiTagDataDispatcher"/>
```

Simple Access Operations

Tag Access operations can be performed on a specific tag or can be applied on tags that match a specific Access-Filter. If no Access-Filter is specified, the Access Operation is performed on all tags in the field of view of chosen antennas. This section covers the Simple Tag Access operation on a specific tag which could be in the field of view of any of the antennas of the connected RFID reader.

Dynamic power optimization should be disabled before any access operations.

```
// Set Dynamic power state off
reader.Configurations.DynamicPowerConfig.setDPOState(DYNAMIC_POWER_OPTIMIZATION.DISABLE);
```

Read

The application can call method **reader.AccessOperations.TagRead.Read()** to read data from a specific memory bank.

```
// Set memory bank for tag read
reader.AccessOperations.TagRead.MemoryBank = MEMORY_BANK.EPC;

// Set tag read length
reader.AccessOperations.TagRead.Length = 0;
```

```
// Set the tag read offset
reader.AccessOperations.TagRead.Offset = 2;

// Set the tag read password
reader.AccessOperations.TagRead.Password = "00";

// Set tag pattern by converting the tag pattern to byte array
int len = TagPattern.Length;
byte[] data = new byte[len / 2];
for (int i = 0; i < len; i += 2)
{
        data[i / 2] = Convert.ToByte(TagPattern.Substring(i, 2), 16);
}
reader.AccessOperations.TagRead.TagPattern = data;

// Read Tag
reader.AccessOperations.TagRead.Read();
```

Write

The application can call method **reader.AccessOperations.TagWrite.Write()** to write data to a specific memory bank. The response is returned as a Tagdata from where number of words can be retrieved.

```
// Set memory bank for tag write
reader.AccessOperations.TagWrite.MemoryBank = MEMORY_BANK.EPC;

// Set the tag write offset
reader.AccessOperations.TagWrite.Offset = 2;

// Set the tag write password
reader.AccessOperations.TagWrite.Password = "00";

// Set tag pattern by converting the tag pattern to byte array
int len = TagPattern.Length;
byte[] data = new byte[len / 2];
for (int i = 0; i < len; i += 2)
{
        data[i / 2] = Convert.ToByte(TagPattern.Substring(i, 2), 16);
}
reader.AccessOperations.TagWrite.TagPattern = data;

// Set tag data to write
len = TagDataToWrite.Length;
byte[] dataToWrite = new byte[len / 2];
for (int i = 0; i < len; i += 2)
{
        dataToWrite[i / 2] = Convert.ToByte(TagDataToWrite.Substring(i, 2), 16);
}
reader.AccessOperations.TagWrite.TagData = dataToWrite;
// Write tag
reader.AccessOperations.TagWrite.Write();
```

Lock

The application can call method **reader.AccessOperations.TagLock.Lock()** to perform a lock operation on one or more memory banks with specific privileges.

```
// Set memory bank for tag lock
```

```
reader.AccessOperations.TagLock.MemoryBank = LOCK_MEMORY_BANK.EPC;

// Set tag lock permission
reader.AccessOperations.TagLock.Permission = ACCESS_PERMISSIONS.READ_WRITE;

// Set the tag lock password
reader.AccessOperations.TagLock.Password = "00";

// Set tag pattern by converting the tag pattern to byte array
int len = TagPattern.Length;
byte[] data = new byte[len / 2];
for (int i = 0; i < len; i += 2)
{
        data[i / 2] = Convert.ToByte(TagPattern.Substring(i, 2), 16);
}
reader.AccessOperations.TagLock.TagPattern = data;
//Lock Tag
reader.AccessOperations.TagLock.Lock();
```

Kill

The application can call method **reader.AccessOperations.TagKill.Kill()** to kill a tag.

```
// Set the tag kill password
reader.AccessOperations.TagKill.Password = "00";

// Set tag pattern by converting the tag pattern to byte array
int len = TagPattern.Length;
byte[] data = new byte[len / 2];
for (int i = 0; i < len; i += 2)
{
        data[i / 2] = Convert.ToByte(TagPattern.Substring(i, 2), 16);
}
reader.AccessOperations.TagKill.TagPattern = data;

// Kill Tag
reader.AccessOperations.TagKill.Kill();
```

Tag Locationing

This feature is supported only on hand-held readers and is useful to locate a specific tag in the field of view of the reader's antenna. The default locationing algorithm supported on the reader can perform locationing only on a single antenna.

reader.TagLocate.Perform(string epc) can be used to start locating a tag, and **reader.TagLocate.Stop()** to stop the locationing operation. The result of locationing of a tag is reported as **reader.TagLocate.ProximityPercentReceived** event and **ProximityPercent** in **ProximityPercentReceivedEventArgs** gives the relative distance of the tag from the reader antenna.

Advance Operations

Using Triggers

Triggers are the conditions that should be satisfied to start or stop an operation (Inventory). This information can be specified using **TriggerInfo** class.

Use **reader.Configurations.TriggerInfo.StartTrigger** and **reader.Configurations.TriggerInfo.StopTrigger** methods to set triggers on the reader.

The following are some use-cases of using TRIGGER_INFO:

- Periodic Inventory: Start inventory at a specified time for a specified duration repeatedly.

```
TriggerInfo triggerInfo = reader.Configurations.TriggerInfo;
// Start inventory every 3 seconds
triggerInfo.StartTrigger.Type=START_TRIGGER_TYPE.START_TRIGGER_TYPE_PERIODIC;
triggerInfo.StartTrigger.Periodic.Period=3000
// Stop trigger
triggerInfo.StopTrigger.Type=STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_DURATION;
triggerInfo.StopTrigger.Duration=5000; // stop after 5 seconds
reader.Configurations.TriggerInfo = triggerInfo;
```

- Perform 'n' Rounds of Inventory with a timeout: Start condition could be any; Stop condition is to perform 'n' rounds of inventory and then stop or stop inventory after the specified timeout.

```
TriggerInfo triggerInfo = reader.Configurations.TriggerInfo;
// Start inventory immediate
triggerInfo.StartTrigger.Type=START_TRIGGER_TYPE.START_TRIGGER_TYPE_IMMEDIATE;
// Stop trigger
triggerInfo.StopTrigger.Type=STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_N_ATTEMPTS_WITH_TIMEOUT;
triggerInfo.StopTrigger.NumAttempts.N=3; // perform 3 rounds of inventory
triggerInfo.StopTrigger.NumAttempts.Timeout=3000; // timeout after 3 seconds
reader.Configurations.TriggerInfo = triggerInfo;
```

- Read 'n' tags with a timeout: Start condition could be any; Stop condition is to stop after reading 'n' tags or stop inventory after the specified timeout.

```
TriggerInfo triggerInfo = reader.Configurations.TriggerInfo;
// Start inventory immediate
triggerInfo.StartTrigger.Type = START_TRIGGER_TYPE.START_TRIGGER_TYPE_IMMEDIATE;
// Stop trigger
triggerInfo.StopTrigger.Type = STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_TAG_OBSERVATION_WITH_TIMEOUT;
triggerInfo.StopTrigger.TagObservation.N= 5; // number of tag observations
triggerInfo.StopTrigger.TagObservation.Timeout = 10000; // timeout after 10 seconds
reader.Configurations.TriggerInfo = triggerInfo;
```

- Inventory based on hand-held trigger: Start inventory when the reader hand-held trigger button is pulled, and stop inventory when the hand-held trigger button is released or subject to timeout.

```
TriggerInfo triggerInfo = reader.Configurations.TriggerInfo;
// Start inventory immediate
triggerInfo.StartTrigger.Type = START_TRIGGER_TYPE.START_TRIGGER_TYPE_HANDHELD;
triggerInfo.StartTrigger.Handheld.HandheldEvent =
      HANDHELD_TRIGGER_EVENT_TYPE.HANDHELD_TRIGGER_PRESSED; // number of tag observations
triggerInfo.StartTrigger.Handheld.Timeout = 10000; // timeout after 10 seconds
// Stop trigger
triggerInfo.StopTrigger.Type = STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_HANDHELD_WITH_TIMEOUT;
triggerInfo.StopTrigger.Handheld.HandheldEvent=
    HANDHELD_TRIGGER_EVENT_TYPE.HANDHELD_TRIGGER_RELEASED; // number of tag observations
triggerInfo.StopTrigger.Handheld.Timeout = 10000; // timeout after 10 seconds

reader.Configurations.TriggerInfo = triggerInfo;
```

Using Beeper

Use the **reader.Configurations.BeeperVolume** property to turn the beeper on/off, and set volume.

Get beeper setting example:

```
BEEPER_VOLUME beeperVolume = reader.Configurations.BeeperVolume;
string strBeeperVolume = "";
switch (beeperVolume)
{
    case BEEPER_VOLUME.HIGH_BEEP:
            strBeeperVolume = "HIGH_BEEP";
            break;
    case BEEPER_VOLUME.MEDIUM_BEEP:
            strBeeperVolume = "MEDIUM_BEEP";
            break;
    case BEEPER_VOLUME.LOW_BEEP:
            strBeeperVolume = "LOW_BEEP";
            break;
    case BEEPER_VOLUME.QUIET_BEEP:  // beeper sound off
            strBeeperVolume = "QUIET_BEEP";
            break;
}
Console.WriteLine("GetBeeperVolume = " + strBeeperVolume);
```

Set beeper example:

```
// Set beeper volume high
reader.Configurations.BeeperVolume = BEEPER_VOLUME.HIGH_BEEP;
```

Batch Mode

When the RFID reader is configured to operate in batch mode, it is capable of reading RFID tag data without being connected to a host device.

The **reader.Configurations.BatchModeConfig** property can be used to configure Batch Mode as follows :

```
reader.Configurations.BatchModeConfig = BATCH_MODE.ENABLE
```

Batch mode can be configured to one of the following:

MODE	Description
BATCH_MODE.DISABLE	Tags are reported in real time as they are inventoried. No data is preserved if the application disconnects.
BATCH_MODE.ENABLE	Tags are stored in an internal database maintained in the reader, and are not returned to host in real time. While in batch mode, the reader will continue to perform inventory even if the reader is disconnected from the host. Upon re-connection, the ReadSessionBatchModeEventArgs" event will be raised indicating that the inventory is in progress. In order to retrieve the stored tags, inventory must be stopped by

	calling reader.Inventory.Stop(), and the reader.Inventory.GetBatchedTags() method must be called to get the stored tag Data.
BATCH_MODE.AUTO	Tags are reported in real time while the application that initiated performing inventory is still connected. If the reader is disconnected, the tag data is stored in an internal database maintained in the reader Upon re-connection, the "ReadSessionBatchModeEventArgs" event will be raised indicating that the inventory is in progress. In order to retrieve the stored tags, inventory must be stopped by calling reader.Inventory.Stop(), and the reader.Inventory.GetBatchedTags() method must be called to get the stored tag Data.

To clear stored batched tags in the reader's internal database, call the **reader.Inventory.PurgeTags()** method.

Using Pre-Filters

Pre-filters are the same as the Select command of C1G2 specification. Once applied, pre-filters are applied prior to Inventory and Access operations.

Singulation

Singulation refers to the method of identifying an individual Tag in a multiple-Tag environment.

To filter tags that match a specific condition, it is necessary to use the tag-sessions and their states (setting the tags to different states based on match criteria –

reader.PreFilters.ConfiguredFilters) so that while performing inventory, tags can be instructed to participate (singulation – **reader.Config.PreFilters.ConfiguredFilters[filterIndex].IsEnable**) or not participate in the inventory based on their states.

Sessions and Inventoried Flags

Tags provide four sessions (denoted S0, S1, S2, and S3) and maintain an independent inventoried flag for each session. Each of the four inventoried flags has two values, denoted A and B. These inventoried flag of each session can be set to A or B based on match criteria using method: **reader.ConfiguredFilters[filterIndex].Action**

Selected Flag

Tags provide a selected flag, SL, which can be asserted or deasserted based on match criteria using: **reader.ConfiguredFilters[filterIndex].Action**

State-Aware Singulation

In state-aware singulation the application can specify detailed controls for singulation: Action and Target.

Action indicates whether matching Tags assert or deassert SL (Selected Flag), or set their inventoried flag to A or to B. Tags conforming to the match criteria specified using the **reader.ConfiguredFilters[filterIndex].Action** are considered matching and the remaining are non-matching.

Target indicates whether to modify a tag's SL flag or its inventoried flag, and in the case of inventoried it further specifies one of four sessions.

Applying Pre-Filters

The following are the steps to use pre-filters:

- Add pre-filters
- Set appropriate singulation controls
- Perform Inventory or Access operation

Add Pre-Filters

The application can update pre-filters using **reader.ConfiguredFilters** list to add and remove Pre-Filters

Set Appropriate Singulation Controls

Now that the pre-filters are set (i.e., tags are classified into matching or non-matching criteria), the application needs to specify which tags should participate in the inventory using **reader.Configurations.Antennas[curAntennaID].SingulationControl**.

Perform Inventory or Access operation

Inventory or Access operation when performed after setting pre-filters, use the tags filtered out of pre-filters for their operation.

RFID Reader Key-Remapping

Key-Remapping enables upper and lower trigger configuration to one of the following options:

1) RFID
2) Sled scanner
3) Terminal scanner
4) Scan notification
5) No action

Retrieve RFID reader trigger mapping information as follows:

```
TriggerMapping triggerMapping= reader.Configurations.TriggerMapping;
Log($"Upper Trigger : {triggerMapping.UpperTrigger}");
Log($"Lower Trigger : {triggerMapping.LowerTrigger}");
```

To set RFID reader trigger mapping (example: set upper trigger to RFID, and lower trigger to sled scan) :

```
TriggerMapping triggerMapping = new TriggerMapping();
triggerMapping.UpperTrigger = TRIGGER_MAPPING.RFID;
triggerMapping.LowerTrigger = TRIGGER_MAPPING.SLED_SCANNER;

reader.Configurations.TriggerMapping = triggerMapping;
```

RSM Attribute Configuration

There are several RSM (Remote Scanner Management) attributes/settings that are configurable in the reader.

Setting a RSM Attribute

To set a RSM attribute call the following command:

```
int attributeId = 1;
string attribType = "F";
string attribValue = "1"; // True
reader.RemoteScannerManagement.SetAttribute(attributeId, attribType, attribValue);
```

Getting a RSM Attribute

To retrieve a RSM attribute:

```
int attributeId = 1; // Attribute ID to get
RsmAttribute attribute= reader.RemoteScannerManagement.GetAttribute(attributeId);
Log($"Attribute ID: {attribute.Number}");
Log($"Attribute Type: {attribute.Type}");
Log($"Attribute Value: {attribute.Value}");
```

Exceptions

The Zebra RFID Windows SDK throws standard .Net exceptions. All API calls should be under try-catch block to catch exceptions thrown while performing API calls.

```
try
{
    // Establish connection to the RFID Reader
    reader.Connect();
}
catch (Exception e)
{
    Debug.Print(e.Message);
```

RFID SDK Demo Application

The Desktop Windows RFID SDK Sample Application shows how to call the RFID Windows API to communicate/configure the RFID reader and receive tag data.

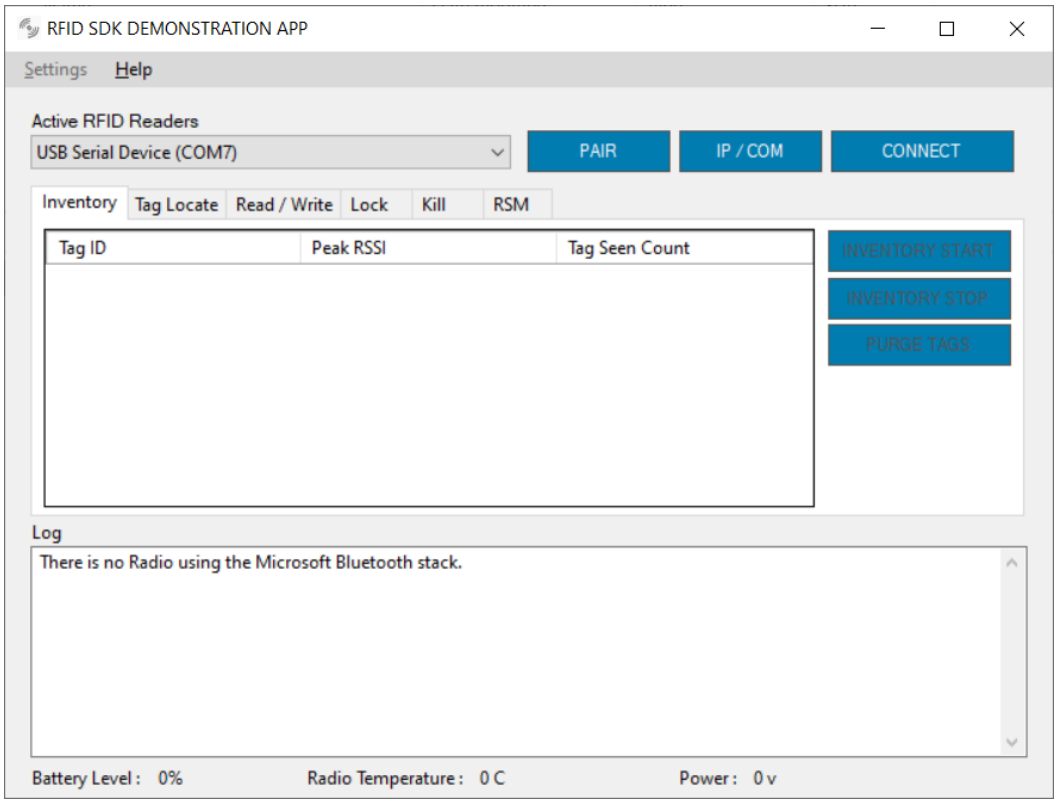

Figure 8-1 *Windows SDK RFID Demo Application*