Octane SDK C# Workbook

REV 1.0 2010-12-14

Proprietary and Confidential

www.impinj.com

Contents

Figures

Introduction

The Octane SDK includes the core library that increases the usability of the Impinj Reader by simplifying the creation of Reader applications. It does this by acting as a wrapper for extracting, modifying, and the application of a Reader's Low Level Reader Protocol (LLRP) settings. This provides high-level control over Reader settings, tag query, and tag-write operations which reduces the effort necessary to manipulate Reader functionality.

Font Group	Example Style
Code command	`Example 1 Reader =>`
File names	*Octanesdk.zip*

Example code	`using Impinj.OctaneSdk;`
	`namespace QueryReaderSettings`

Key Features

The core library is delivered in multiple programming languages and is completed by a variety of documentation. On-Reader applications (C++ only) are written in the same manner as off-Reader applications, and all programming languages have a consistent implementation. The documentation is available in traditional online help *.chm* files, and also includes examples and templates:

- Speedway Reader API
- C#, C++ Linux
- Examples, Templates, and Online Help

Getting Started with the Octane SDK

The Octane SDK is delivered in binary form as a DLL. Sample content is delivered as source code. Before the exercises in this workbook can be accomplished, the SDK must be unzipped. Visual Studio 2010 (VS2010) C# Express must be installed.

Pre Workshop Tasks

Task 1 Computer and Reader

1. Acquire a computer and Revolution Reader, and then network them together.
2. To complete the tag access exercise, a Reader antenna and a few tags are required. Use Monza 4 tags if available, but any EPC GEN 2 type will work.
3. Connect the GPIO board or GPIO Connector Box to the multi-purpose DE15 connector, and connect the Reader antenna to port 1 as shown in Figure 1 below.

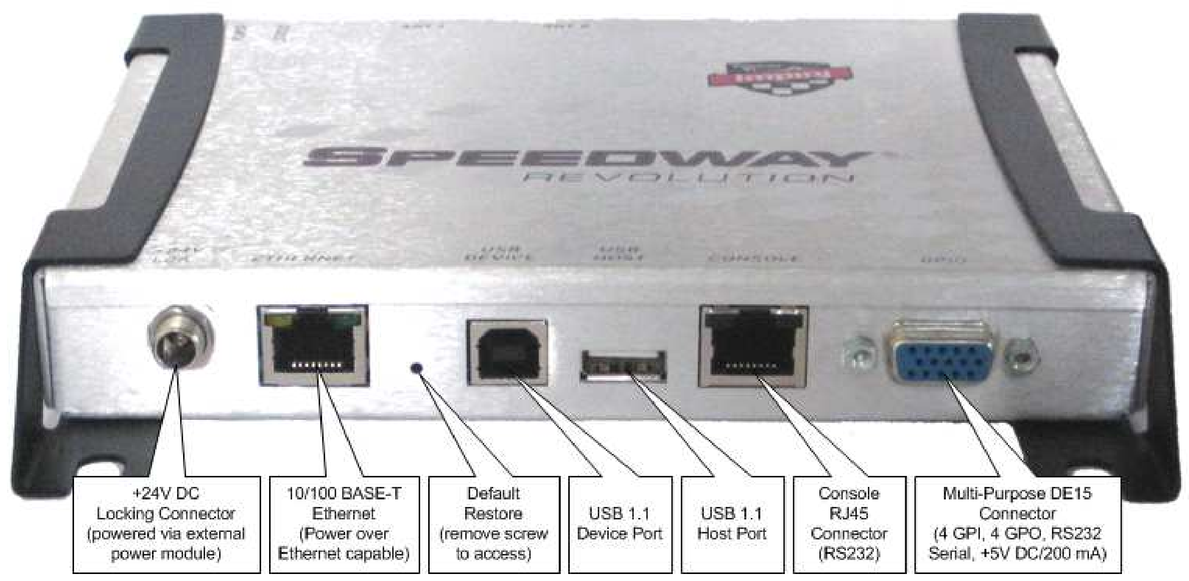

+24V DC Locking Connector (powered via external power module)
10/100 BASE-T Ethernet (Power over Ethernet capable)
Default Restore (remove screw to access)
USB 1.1 Device Port
USB 1.1 Host Port
Console RJ45 Connector (RS232)
Multi-Purpose DE15 Connector (4 GPI, 4 GPO, RS232 Serial, +5V DC/200 mA)
SPEEDWAY REVOLUTION

Figure 1 Speedway Reader Connections

Task 2–Unzip SDK

1. *Octanesdk.zip* contains the libraries, technical reference (Help text), and this Workbook.
2. Unzip the file to a local directory.

Note: This tutorial is included with the *OctaneSdk.* If you update to a new *OctaneSdk* version, check the *OctaneSdk* web page for a current version of this document.

Task 3 – Install Visual C# 2010 Express

To install VS2010 C# Express, complete the following steps:

1. Download and run the VS2010 installer, here: http://www.microsoft.com/express/downloads/#2010-Visual-CS .
2. Restart your computer to complete the installation process. .
3. Launch VS2010 and allow it to complete the one-time initialization process. This should take about 2 to 3 minutes.

Task 4 – Exercise Examples

1. From VS2010, select File, then Open Project.
2. Select the */octane/cs/sdk/OctaneSdk.sln*. file.
3. Complete the file conversion. (This sample file was built using VS 2008.)

4. Right click on the desired program file (*xxxxx.cs*) and then select View Code to view the code for each project. See Figure 3.

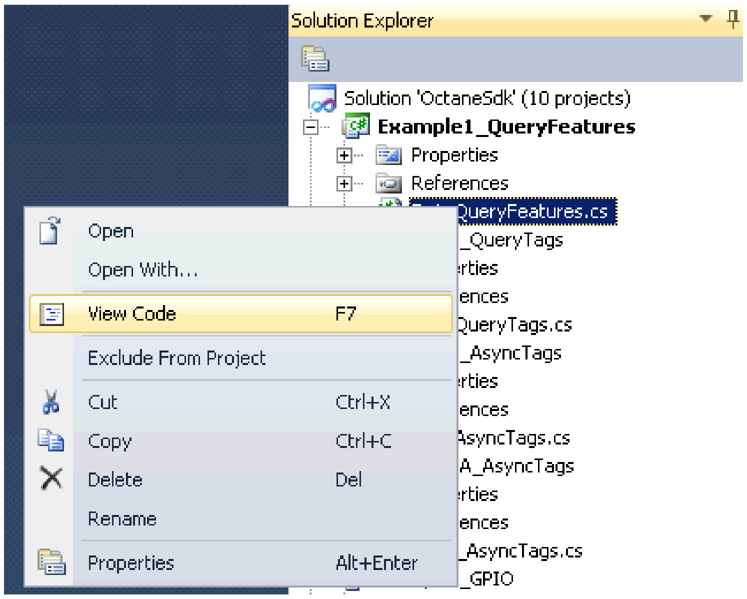


Figure 2 Viewing Codin in C# OctaneSdk Sample

5. In the Solution Explorer, right click on ***Example1_QueryFeatures*** and then select Set As StartUp Project.
6. Select Debug, and then Start Debugging. Enter your Reader's name at the prompt and press enter.
7. When done reviewing the Reader's details, press any key to exit.

```
prompt:
  Example 1 Reader => {your reader's name or IP address}
output:
  Model               Speedway R220
  Software Version    4.4.1.3
  Firmware Version    4.4.0.17
  PCBA Version        270-001-003
  FPGA Version        4.4.0.240
  Regulator Region    ETSI_EN_302_208_v1_2_1
  Antennas            1:Connected 2:Unconnected 3:N/A 4:N/A
prompt:
  Done => {press any key}
```

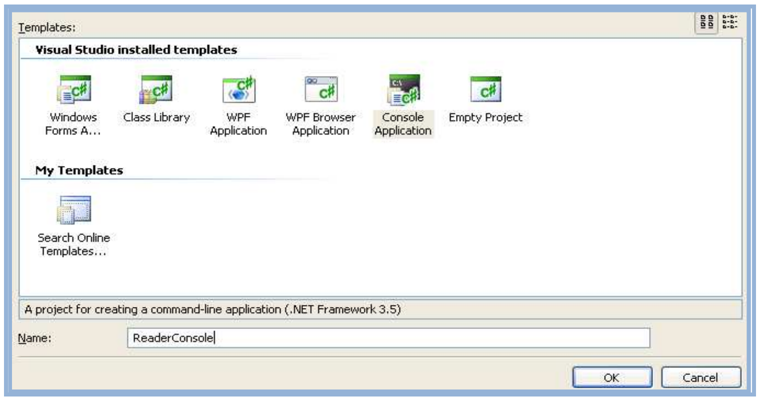
Figure 1
Exercise Example 1

Note: The output from your Reader will be different from the above example, depending on the Reader model, region, and version information.

8. Build and run the other examples to increase your skill using the SDK. Right click the Example project, select Set As StartUp Project, and then Debug, and finally Start Debugging.

Task 5 Preparing for Exercises

1. For each exercise, create a new project by clicking File, and then New Project.
2. Select Console Application as the template type in the New Project dialog
3. Name the New Project something easy to remember. By default the project will be created in My Documents/Visual Studio 2010/Projects. See Figure 4.The project will be created in its own folder.



A project for creating a command-line application (.NET Framework 3.5)

Name: ReaderConsole

Figure 3 Visual Studio New Project Dialog Box

3. Select ***Solution Explorer*** and right click on the References folder, and then select Add Reference. Each time a project is created, the SDK libraries must be imported into the new project.

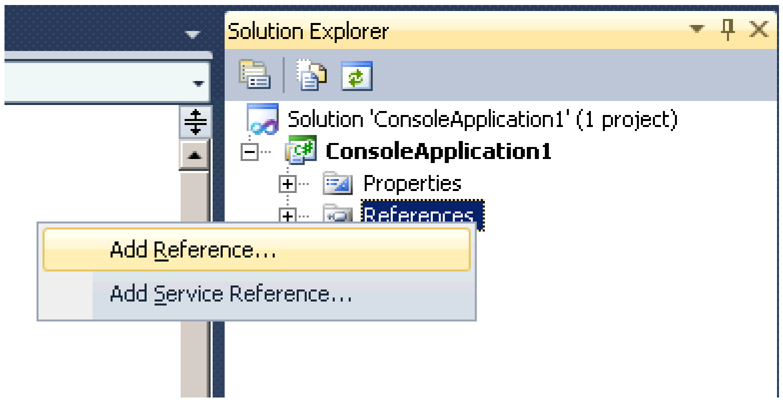


Figure 4 Visual Studio Adding References for OctandSdk

4. Browse to */OctaneSdk/Libraries/Debug*, and then select all three DLLs. See Figure 6.

5. Click OK.

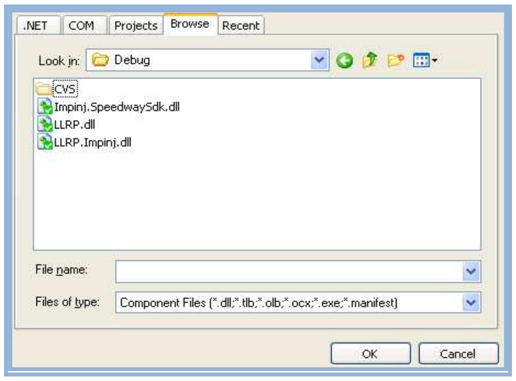
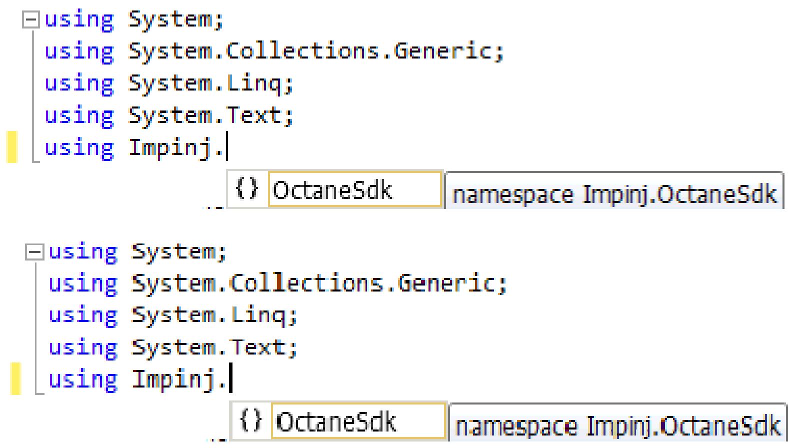


Figure 5 Visual Studio Add Reference Dialog Box

6. Type "*using Impinj.OctaneSdk*" at the top of the *Program.cs* file. Note that the Visual Studio Intellisense helps with this (use Tab button to complete auto fill).



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.
```

Figure 6 Adding Impinj.OctaneSdk Reference to C#.NET Program.cs

7. Select File and then Save All.

The new project is now prepared for the exercises.

Settings

Speedway Readers have a wide range of capabilities. With so many capabilities, it requires configuration to reduce them to a manageable subset. The capabilities can be thought of as a programming language, or a set of tools waiting for a purpose. That purpose is defined by Reader settings. The settings are a program that describes what capabilities are used and how they behave. That means that the operation of Reader is critically dependent on the proper settings. The application depends upon the Reader. As a programmer, you must understand the settings and how they affect the Reader's operation.

The easiest way to get guaranteed success changing a Reader's settings is to have the library catalog a default, or "best guess" settings object. Once that object is returned, the Reader settings can be changed with the default. Settings can be modified in code, written to disk, edited, and loaded back in. Settings are expressed as XML on disk, and also as a `class`.

Query Services

Name	Description
QueryFactorySettings	Determines most basic settings from the Reader's capabilities and features.
QuerySettings	If the settings were changed, this returns current settings configuration.
QueryFeatureSet	Definition of features that are available on the Reader.
QueryStatus	The current state of the Reader, antennas,and more.

Command Services

Name	Description
Connect	Connects to a Reader allowing query and command services to transact.
Disconnect	After disconnect, no services are available until next connect.
ClearSettings	Deletes the Readers settings and returns to the factory default settings.
ApplySettings	Replaces theReader's current settings with the passed-in settings.

Query, Save, and Load Settings

In the following exercise you will connect to the Reader, save settings to an XML file, then edit and load the XML file to apply changes to the Reader settings.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace QueryReaderSettings
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void DisplayCurrentSettings()
        {
            // Query the current reader settings and print the results.
            Console.WriteLine("Reader Settings");
            Console.WriteLine("---------------");

            Settings settings = Reader.QuerySettings();
            Console.WriteLine("Reader mode : {0}", settings.ReaderMode);
            Console.WriteLine("Search mode : {0}", settings.SearchMode);
            Console.WriteLine("Session : {0}", settings.Session);
            Console.WriteLine("Rx sensitivity (Antenna 1) : {0} dBm",
                settings.Antennas[1].RxSensitivityInDbm);
            Console.WriteLine("Tx power (Antenna 1) : {0} dBm",
                settings.Antennas[1].TxPowerInDbm);

            Console.WriteLine("");
        }

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Query the reader features and print the results.
                Console.WriteLine("Reader Features");
                Console.WriteLine("---------------");
                FeatureSet features = Reader.QueryFeatureSet();
                Console.WriteLine("Model name : {0}", features.ModelName);
                Console.WriteLine("Model number : {0}", features.ModelNumber);
                Console.WriteLine("Reader identity : {0}", features.ReaderIdentity);
                Console.WriteLine("Firmware version : {0}",
                    features.FirmwareVersion);
                Console.WriteLine("Antenna count : {0}\n", features.AntennaCount);


                // Write the reader features to file.
                features.Save("features.xml");
                // Query the current reader status.
                Console.WriteLine("Reader Status");
                Console.WriteLine("---------------");
                Status status = Reader.QueryStatus();
                Console.WriteLine("Is connected : {0}", status.IsConnected);
                Console.WriteLine("Is singulating : {0}", status.IsSingulating);
                Console.WriteLine("Temperature : {0} degrees\n",
                    status.TemperatureInCelsius);

                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();

                // Display the current reader settings.
                DisplayCurrentSettings();

                // Save the settings to file in XML format.
                Console.WriteLine("Saving settings to file.");
                Settings settings = Reader.QuerySettings();
                settings.Save("settings.xml");

                // Wait here, so we can edit the
                // settings.xml file in a text editor.
                Console.WriteLine("Edit settings.xml and press enter.");
                Console.ReadLine();

                // Load the modified settings from file.
                Console.WriteLine("Loading settings from file.");
                settings = Settings.Load("settings.xml");

                // Apply the settings we just loaded from file.
                Console.WriteLine("Applying settings from file.\n");
                Reader.ApplySettings(settings);

                // Display the settings again to show the changes.
                DisplayCurrentSettings();

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception : {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

There are three memory banks on an EPC GEN 2 tag that you need to know how to query: the Electronic Product Code (EPC), User Memory, and Tag Identification (TID).

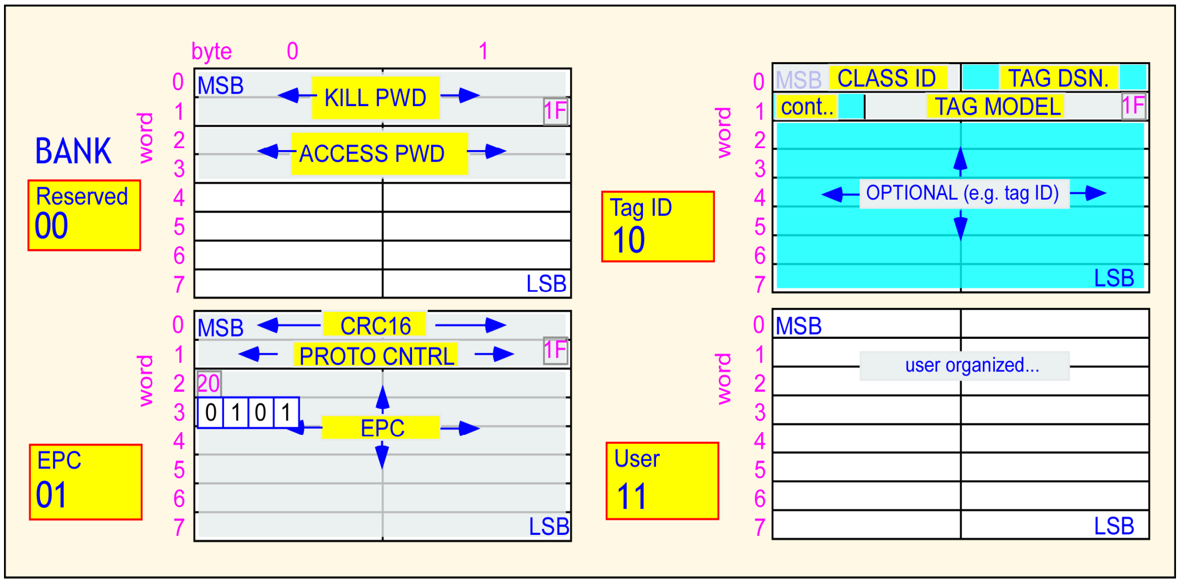


Figure 7 EPC Gen 2 tag memory banks including EPC (1, or 01 in binary), TID (2, or 10) and User (3 or 11).

Note: the data locations in figure 8. Focus attention on the EPC. The actual EPC begins at word 2 (or hex address 20). Prior to the EPC is the 16 bit Cyclic Redundancy Check or CRC (word 0) and the Protocol Control (or PC) word (word 1).

The exercises in this section enable you to read the three memory banks using two different approaches: synchronously and asynchronously. You will also apply the feature sets of the SDK such as filters where only tags with specific data will respond and serialized TID (tag returns EPC and TID together using one command). Figure 8 shows the memory available for the various version of the Impinj Monza 4 tag IC.

Model	User Memory	EPC Memory	True3D Antenna Technology	Serialized TID	QT technology
Monza 4QT	512	128	✓	✓	✓
Monza 4E	128	496	✓	✓	–
Monza 4D	32	128	✓	✓	–

Figure 8 Memory and features available on Monza 4 models

Read Tags Synchronously

In this exercise, you will configure the Reader so that observed tag data is stored in Reader memory and a report of all observed tags is sent only when commanded by the client application.

```
using System;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadTagsSync
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Wait until the tag query has ended
                // before sending the tag report.
                settings.Report.Mode = ReportMode.WaitForQuery;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Read tags for 5 seconds
                Console.WriteLine("Reading tags...");
                TagReport tagReport = Reader.QueryTags(5);




                // Print out the results.
                foreach (Tag tag in tagReport.Tags)
                {
                    Console.WriteLine("EPC : {0} Antenna : {1}",
                                                tag.Epc, tag.AntennaPortNumber);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Read Tags Asynchronously

In this exercise, you will configure the reader to report each tag to the client application as soon as it is observed. This is known as “asynchronous reporting”.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadTagsAsync
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Assign the TagsReported handler.
                // This specifies which function to call
                // when tags reports are available.
                Reader.TagsReported += new EventHandler
                    <TagsReportedEventArgs>(OnTagsReported);

                // Start reading.
                Reader.Start();

                // Wait for the user to press enter.
                Console.WriteLine("Press enter when done.");
                Console.ReadLine();

                // Stop reading.
                Reader.Stop();

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
        }

        static void OnTagsReported(object sender, TagsReportedEventArgs args)
        {
            // This function is called asynchronously
            // when tag reports are available.
            // Loop through each tag in the report
            // and print the data.
            foreach (Tag tag in args.TagReport.Tags)
            {
                Console.WriteLine("EPC : {0} Antenna : {1}",
                                    tag.Epc, tag.AntennaPortNumber);
            }
        }
    }
}
```

Read Tags Using Periodic Trigger

There may be cases where you want the reader to conduct ‘polling’ or examination for tags. During polling, the reader initiates a scan for tags for a specified period of time and then waits for a set period before scanning again. An example of this would be a “smart-shelf” application where the user does not need to know the instant that a tagged item is placed on, or removed from the shelf. Updating polling every 10 seconds is sufficient and reduces both network and RF congestion.

There are a number of trigger types and settings available: refer to the SDK Quick Reference Guide or the SDK Compiled HTML Help File for more information.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadTagsPeriodicTrigger
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Reading tags for 5 seconds every 10 seconds
                settings.AutoStart.Mode = AutoStartMode.Periodic;
                settings.AutoStart.PeriodInMs = 10000;
                settings.AutoStop.Mode = AutoStopMode.Duration;
                settings.AutoStop.DurationInMs = 5000;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);
                // Assign the TagsReported handler.
                // This specifies which function to call
                // when tags reports are available.
                Reader.TagsReported += new EventHandler
                    <TagsReportedEventArgs>(OnTagsReported);

                // Wait for the user to press enter.
                Console.WriteLine("Press enter when done.");
                Console.ReadLine();

                // Stop reading.
                Reader.Stop();

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
        }

        static void OnTagsReported(object sender, TagsReportedEventArgs args)
        {
            // This function is called asynchronously
            // when tag reports are available.
            // Loop through each tag in the report
            // and print the data.
            foreach (Tag tag in args.TagReport.Tags)
            {
                Console.WriteLine("EPC : {0} Antenna : {1}",
                                    tag.Epc, tag.AntennaPortNumber);
            }
        }
    }
}
```

Read Tags Using GEN 2 Filter

The Impinj Revolution Reader and Monza tags support what is known as GEN 2 filtering. There are two possible approaches to take when you only want to read certain tags based on their data. You can either use either EPC, User, TID, or a combination of these. One is to have all the tags in the read- zone and backscatter their data to the reader, then use filtering in the application layer (your program).

The other `method`, which you will utilize here, is to configure the reader so that it commands only tags matching the filter to respond while the others will stay silent. This is useful when applied in the physical layer: only tags matching the filter will backscatter data and it can decrease the amount of RF and tag-to-tag interference.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadTagsFiltered
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Setup a tag filter.
                // Only the tags that match this filter will respond.
                // We're only going to use filter #1.
                settings.Filters.Mode = TagFilterMode.OnlyFilter1;
                // We want to apply the filter to the EPC memory bank.
                settings.Filters.TagFilter1.MemoryBank = MemoryBank.Epc;
                // Start matching at address 0x20, since the
                // first 32-bits of the EPC memory bank are the
                // CRC and control bits.
                settings.Filters.TagFilter1.BitPointer = 0x20;
                // Our filter is 16-bits long (the first word of the EPC).
                settings.Filters.TagFilter1.BitCount = 16;
                // Only match tags with EPCs that start with "3008"
                settings.Filters.TagFilter1.TagMask = "3008";
                // We want to include tags that match this filter.
                // Alternatively, we could exclude tags that
                // match the filter.
                settings.Filters.TagFilter1.FilterOp = TagFilterOp.Match;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);
                // Assign the TagsReported handler.
                // This specifies which function to call
                // when tags reports are available.
                Reader.TagsReported += new EventHandler
                    <TagsReportedEventArgs>(OnTagsReported);
                // Start reading.
                Reader.Start();
                // Wait for the user to press enter.
                Console.WriteLine("Press enter when done.");
                Console.ReadLine();
                // Stop reading.
                Reader.Stop();
                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
        }

        static void OnTagsReported(object sender, TagsReportedEventArgs args)
        {
            // This function is called asynchronously
            // when tag reports are available.
            // Loop through each tag in the report
            // and print the data.
            foreach (Tag tag in args.TagReport.Tags)
            {
                Console.WriteLine("EPC : {0} Antenna : {1}",
                                    tag.Epc, tag.AntennaPortNumber);
            }
        }
    }
}
```

Read User Memory

The Impinj Monza 4 tags offer up to 512 bits of user memory. In this exercise you will query the user memory using exception handling best practices.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadUserMemory
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();

                // Define how we want to perform the read.
        ReadUserMemoryParams readParams = new ReadUserMemoryParams();
                // Use antenna #1.
                readParams.AntennaPortNumber = 1;
                // No access password required for this tag.
                readParams.AccessPassword = null;
                // Start reading from the base of user memory (address 0).
                readParams.WordPointer = 0;
                // Read 32 words of user memory (512-bits).
                readParams.WordCount = 32;
                // Read the first tag we see.
                // Alternatively, we could choose a specific
                // tag by EPC or other identifier.
                readParams.TargetTag = null;
                // Timeout in 5 seconds if the read operation fails.
                readParams.TimeoutInMs = 5000;


               // Perform the read and check the results.
                ReadUserMemoryResult result = Reader.ReadUserMemory(readParams);

                if (result.ReadResult.Result == AccessResult.Success)
                {
                    Console.WriteLine("Tag read successful. Memory contents : {0}",
                        result.ReadResult.ReadData);
                }
                else
                {
                    Console.WriteLine("Error reading tag : {0}",
                        result.ReadResult.Result);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Read Serialized TID

The Impinj Monza 4 tags support the ability to backscatter both their EPC and TID data at the same time. This feature is valuable for authentication purposes. The Monza 4 TID is programmed with a unique, 96-bit number, and perm-write locked at the factory. In this exercise you will read the TID by copying the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadSerializedTid
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static string ReadTid(string epc)
        {
            // Read the TID memory bank.
            ReadTidMemoryParams readParams = new ReadTidMemoryParams();
            // No password set.
            readParams.AccessPassword = null;
            // Read from all antennas.
            readParams.AntennaPortNumber = 0;
            // Specify which tag to read by EPC.
            readParams.TargetTag = epc;
            // Read the entire TID bank (two 16-bit words).
            readParams.WordCount = 2;
            // Set the read timeout in milliseconds.
            readParams.TimeoutInMs = 5000;
            // Read the TID memory and return the data.
            ReadTidMemoryResult result = Reader.ReadTidMemory(readParams);
            return result.ReadResult.ReadData;
        }

        static void Main(string[] args)
        {
            string tid;

            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");
                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // and serialized TID (if available)
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;
                settings.Report.IncludeSerializedTid = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Read tags for 5 seconds
                Console.WriteLine("Reading tags...");
                TagReport tagReport = Reader.QueryTags(5);

                // Print out the results.
                foreach (Tag tag in tagReport.Tags)
                {
                    // If this tag supports Serialized TID (Monza4)
                    // then the TID was returned with the tag report.
                    // Otherwise, we must read the TID separately.
                    if (tag.IsSerializedTidPresent)
                    {
                        Console.WriteLine("Read Monza4 tag.");
                        tid = tag.SerializedTid;
                    }
                    else
                    {
                        Console.WriteLine("Read Monza3 tag.");
                        tid = ReadTid(tag.Epc);
                    }
                    Console.WriteLine("EPC : {0} TID : {1} Antenna : {2}",
                                        tag.Epc, tid, tag.AntennaPortNumber);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
```

```
            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Tag Access

Tag access is a general term that covers modifying tag data including writing and locking of EPC, User Memory and passwords, as well as killing a tag. Each tag access command has a three components:

- *method* on the Reader object for each command
- *parameters set* that carries command details
- *result object* specific to the type of command executed

In these exercises, you will program the tag EPC, write to User Memory, and kill or permanently disable the tags.

There are some specific `methods` in Octane SDK for programming Monza tags with Revolution Reader. One is a block-write approach where two hex words are written at a time to the target memory bank instead of word-by-word. Block-write increases tag programming speed. The other is the ability to program EPC, verify success, set the access password, and lock the tag all in one `method` (*ProgramEPCParams*).

Program EPC

The following code shows you how to program an EPC into a tag.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ProgramEpc
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();

                // Define how we want to perform the write.
                ProgramEpcParams epcParams = new ProgramEpcParams();
                // Use antenna #1.
                epcParams.AntennaPortNumber = 1;
                // Write to the first tag we see.
                // Alternatively, we could choose a specific
                // tag by EPC or other identifier.
                epcParams.TargetTag = null;
                // Timeout in 5 seconds if the write operation fails.
                epcParams.TimeoutInMs = 5000;
                // This is the new EPC we will write to the tag.
                epcParams.NewEpc = "0123-4567-89AB-CDEF-0123-4567";

                // Perform the write and check the results.
                ProgramEpcResult result = Reader.ProgramEpc(epcParams);
                if (result.WriteResult.Result == AccessResult.Success)
                {
                    // Show how many words were written to the tag.
                    Console.WriteLine("Tag write successful. {0} words written.",
                        result.WriteResult.NumWordsWritten);
                    // Read back the EPC and print it out.
                    Console.WriteLine("Querying tag...");
                    TagReport report = Reader.QueryTags(2);
                    Console.WriteLine("Tag EPC: {0}", report.Tags[0].Epc);
                }
                else
                {
                    Console.WriteLine("Error writing to tag : {0}",
                        result.WriteResult.Result);
                }
                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Program User Memory

The Monza 4 tag IC supports up to 512 bits (or 32 16-bit words) of user memory. In this exercise you will program a portion of the user memory, the first four words. You could alternately use ProgramUserMemoryParams to program all of a tag's user memory.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ProgramUserMemory
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();

                // Define how we want to perform the write.
                ProgramUserBlockParams writeParams = new ProgramUserBlockParams();
                // Use antenna #1.
                writeParams.AntennaPortNumber = 1;
                // No access password required for this tag.
                writeParams.AccessPassword = null;
                // Start writing at the base of user memory (address 0).
                writeParams.WordPointer = 0;
                // Write to the first tag we see.
                // Alternatively, we could choose a specific
                // tag by EPC or other identifier.
                writeParams.TargetTag = null;
                // Timeout in 5 seconds if the write operation fails.
                writeParams.TimeoutInMs = 5000;
                // Write this data to user memory.
                writeParams.NewUserBlock = "0123456789ABCDEF";

               // Perform the write and check the results.
                ProgramUserBlockResult result =
                    Reader.ProgramUserBlock(writeParams);
                if (result.WriteResult.Result == AccessResult.Success)
                {
                    Console.WriteLine("Tag write successful. {0} words written.",
                        result.WriteResult.NumWordsWritten);
                }
                else
                {
                    Console.WriteLine("Error writing to tag : {0}",
                        result.WriteResult.Result);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Kill Tags

EPC GEN 2 tags allow for a process which completely and permanently disables or “kills” a tag. Once a tag is killed, it becomes effectively useless. It is always good practice to target a specific tag or group of tags to kill and to use unique kill passwords in order to avoid unintentionally disabling a large population of tags in the read-zone (especially if using far-field antennas).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace KillTags
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Set this string to the EPC of the tag you want to kill.
                const string TARGET_TAG = "3008-33B2-DDD9-06C0-0000-1415";

                // Set this string to the new kill password.
                const string NEW_KILL_PW = "12345678";

                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();

                // First, set the kill password
                // since a zero password is not allowed.
                ProgramKillPasswordParams pwParams =
                    new ProgramKillPasswordParams();
                // Use antenna #1.
                pwParams.AntennaPortNumber = 1;
                // No access password required for this tag.
                pwParams.AccessPassword = null;
                // Specify the target tag by EPC.
                pwParams.TargetTag = TARGET_TAG;
                // Timeout in 5 seconds if the operation fails.
                pwParams.TimeoutInMs = 5000;
                // Specify the new kill password (32-bits).
                pwParams.NewKillPassword = NEW_KILL_PW;
                // Program the password and check the result.
                ProgramKillPasswordResult pwResult =
                    Reader.ProgramKillPassword(pwParams);

                if (pwResult.WriteResult.Result == AccessResult.Success)
                {
                    // We've successfully set the kill password.
                    // Now issue the kill command.
                    Console.WriteLine("Successfully set kill password.");
                    KillTagParams killParams = new KillTagParams();
                    // Use antenna #1.
                    killParams.AntennaPortNumber = 1;
                    // Set the kill password.
                    killParams.KillPassword = NEW_KILL_PW;
                    // No access password required for this tag.
                    killParams.AccessPassword = null;
                    // Specify the target tag by EPC.
                    killParams.TargetTag = TARGET_TAG;
                    // Timeout in 5 seconds if the kill operation fails.
                    killParams.TimeoutInMs = 5000;

                    // Perform the kill and check the results.
                    KillTagResult killResult = Reader.KillTag(killParams);
                    if (killResult.KillResult.Result == AccessResult.Success)
                    {
                        Console.WriteLine("Successfully killed tag.");
                    }
                    else
                    {
                        Console.WriteLine("Error killing tag : {0}",
                            killResult.KillResult.Result);
                    }
                }
                else
                {
                    Console.WriteLine("Error setting kill password : {0}",
                        pwResult.WriteResult.Result);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

GPIO

The Reader interacts with the world around it via general purpose inputs (GPI) and general purpose outputs (GPO). These are commonly referred to as GPIO ports. The inputs allow external devices to trigger the reader. Examples include light sensors, motion detectors, pressure mats and more. The outputs allow the reader to affect external devices such as access gates, indicator lights, conveyors and more.

For example, if its desired for the reader to read a tag on an item when it passes by on a conveyor (and not at other times), a GPI could be connected to a light sensor and an OctaneSdk AutoStart trigger could be configured to start the reader when the light sensor changes the GPI to logic high. To stop the reader a timeout could be used. Alternately, an additional light sensor could be attached to a second GPI and an OctaneSdk AutoStop trigger could be configured to stop the reader when this second light sensors changes to the GPI to logic high. Numerous other start and stop configurations are possible. A few are shown in the examples below. GPO could be integrated to offer more visibility and control over the process. When the first GPI is set by the light sensor, the OctaneSdk can be used to set a GPO to an active state turn on a connected light indicator. Similarly, the second GPI could be used to set a GPO to turn off the connected light indicator.

GPIO connectivity is made significantly easier with the use of the Impinj GPIO Connectivity Box. Impinj designed the GPIO Box for use with the Speedway Revolution Reader and provides convenient access to the Reader's GPIO (General Purpose Input/Output) port. The GPIO Box interfaces with the Reader via a supplied HD15 cable, and separates each input and output signal into easy-access screw terminals. Neighboring ground and power terminals accompany each signal terminal to easily connect three-wire devices. The GPIO Box also includes an on-board relay which interfaces with devices that require high wattage or an open/short signal, as well as highly visible LED indicators for each input and output. This simplifies troubleshooting and diagnostics of programming logic.

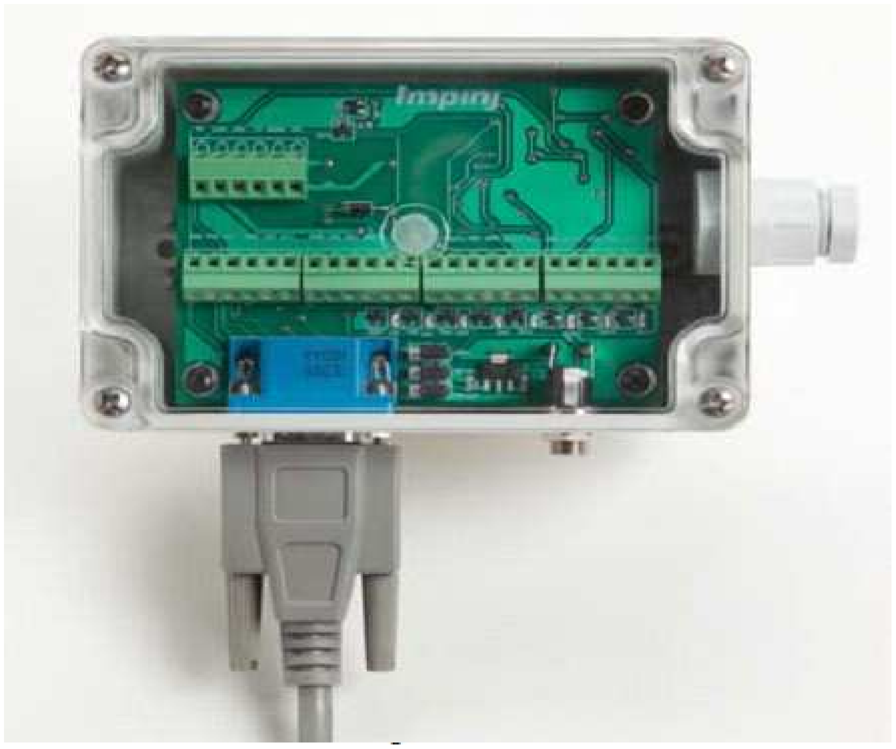


Figure 9 Impinj GPIO Connectivity Box

General Purpose Inputs

This exercise will show you how to use the GPI port to trigger the Reader to start and stop. While this exercise uses the same port, you could easily modify it to have one GPI act as a start trigger and another GPI trigger as a stop trigger as described in the overview scenario.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReadTagsGpiTrigger
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();
                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Start reading tags when GPI #1 goes high.
                settings.Gpis[1].IsEnabled = true;
                settings.Gpis[1].DebounceInMs = 50;
                settings.AutoStart.Mode = AutoStartMode.GpiTrigger;
                settings.AutoStart.GpiPortNumber = 1;
                settings.AutoStart.GpiLevel = true;

                // Stop reading tags when GPI #1 goes low.
                settings.AutoStop.Mode = AutoStopMode.GpiTrigger;
                settings.AutoStop.GpiPortNumber = 1;
                settings.AutoStop.GpiLevel = false;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Assign the TagsReported handler.
                // This specifies which function to call
                // when tags reports are available.
                Reader.TagsReported += new EventHandler
                    <TagsReportedEventArgs>(OnTagsReported);

                // Wait for the user to press enter.
                Console.WriteLine("Press enter when done.");
                Console.ReadLine();

                // Stop reading.
                Reader.Stop();

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
        }
        static void OnTagsReported(object sender, TagsReportedEventArgs args)
        {
            // This function is called asynchronously
            // when tag reports are available.
            // Loop through each tag in the report
            // and print the data.
            foreach (Tag tag in args.TagReport.Tags)
            {
                Console.WriteLine("EPC : {0} Antenna : {1}",
                                    tag.Epc, tag.AntennaPortNumber);
            }
        }
    }
}
```

General Purpose Outputs

This exercise displays how to cycle through and enable each of this GPO ports.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;
using System.Threading;

namespace SetGpos
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();
        static void Main(string[] args)
        {
            int i;

            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");
                // Configure the reader with the factory deafult settings.
                Reader.ApplyFactorySettings();
                // Turn the general purpose ouputs
                // (GPOs) on one at a a time
                Console.WriteLine("Setting general purpose outputs...");
                for (i = 1; i <= 4; i++)
                {
                    Reader.SetGpo(i, true);
                    Thread.Sleep(1500);
                    Reader.SetGpo(i, false);
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Subscribe, Test Power, and Thread

This section contains exercises that cover: subscribing to Reader events, varying Reader power in a loop, and a basic implementation of Windows Presentation Foundation (WPF).

Subscribing to Reader Events

This exercise shows you how to write a host-side program that handles events and maps them to appropriate functionality. In this example, when a GPI trigger occurs or when a tag report is sent to a specific antenna and then triggers an Alert for the user to press Enter. Refer to section 2.4 of the SDK Quick Reference for more details and a complete listing of available event to subscribe to.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace ReaderEvents
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();

        static void Main(string[] args)
        {
            int i;

            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Enable all of the general purpose inputs (GPIs).
                for (i=1; i <= 4; i++)
                {
                    settings.Gpis[i].IsEnabled = true;
                }


              // Enable all of the antenna ports.
                for (i = 1; i <= 4; i++)
                {
                    settings.Antennas[i].IsEnabled = true;
                }

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Assign event handlers for GPI and antenna events.
                Reader.GpiChanged += new
                    EventHandler<GpiChangedEventArgs>(OnGpiEvent);
                Reader.AntennaChanged += new
                    EventHandler<AntennaChangedEventArgs>(OnAntennaEvent);

                // Start the reader (required for antenna events).
                Reader.Start();

                // Wait for the user to press enter.
                Console.WriteLine("Press enter when done.");
                Console.ReadLine();

                // Stop reading.
                Reader.Stop();

                // Disconnect from the reader.
                Reader.Disconnect();

            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }
        }

        // This function gets called when a GPI event occurs.
        static void OnGpiEvent(object sender, GpiChangedEventArgs args)
        {
            Console.WriteLine("A GPI event occurred.");
            Console.WriteLine("Port : {0} State : {1}\n",
                args.PortNumber, args.State);
        }

        // This function gets called when an antenna event occurs.
        static void OnAntennaEvent(object sender, AntennaChangedEventArgs args)
        {
            Console.WriteLine("An antenna event occurred.");
            Console.WriteLine("Port : {0} State : {1}\n",
                args.PortNumber, args.State);
        }
    }
}
```

Power Ramp

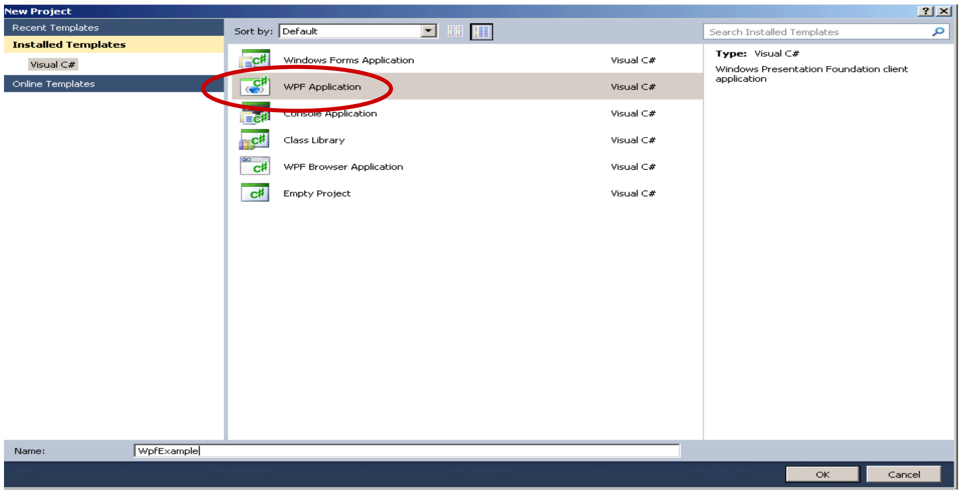
This exercise will enable you to code a power ramp test. The goal is to find the lowest power level that still finds all tags. The transmit power is steadily increased. As the power increases the Reader reports how many tags were singulated at each transmit power level and their RSSI (Return Signal Strength Indicator) .The complete series takes about one minute.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace TxPowerRamp
{
    class Program
    {
        // Create an instance of the SpeedwayReader class.
        static SpeedwayReader Reader = new SpeedwayReader();
        static double minTx, maxTx;

        static void Main(string[] args)
        {
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Query the reader features to get the min and max Tx power.
                FeatureSet features = Reader.QueryFeatureSet();
                minTx = features.TxPowers.Entries.First().Dbm;
                maxTx = features.TxPowers.Entries.Last().Dbm;

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the Peak RSSI
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludePeakRssi = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;



                 // Loop, increasing the transmit power in 1 dBm steps.
                for (double power = minTx; power <= maxTx; power += 1.0)
                {
                    // Set the transmit power (in dBm).
                    settings.Antennas[1].TxPowerInDbm = power;
                    // Apply the new transmit power settings.
                    Reader.ApplySettings(settings);

                    // Read tags for two seconds.
                    TagReport report = Reader.QueryTags(2);

                    // Loop through all the tags in the report
                    // and print out the EPC, Tx Power and Peak RSSI.
                    foreach (Tag tag in report.Tags)
                    {
                        Console.WriteLine("EPC : {0}, Tx Power : {1} dBm,
                                           tag.Epc, power, String.Format("{0:0.00}",
                                           tag.PeakRssiInDbm));
                    }
                }

                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException e)
            {
                Console.WriteLine("Octane SDK exception: {0}", e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception : {0}", e.Message);
            }

            // Wait for the user to press enter.
            Console.WriteLine("Press enter when done.");
            Console.ReadLine();
        }
    }
}
```

Proper Threading Technique using Windows Presentation Format (WPF)

The following section shows how to create a multi-threaded WPF using the Octane SDK.

1. Create a new project by selecting **WPF Application.** See Figure 11.Do not create a Console Application.

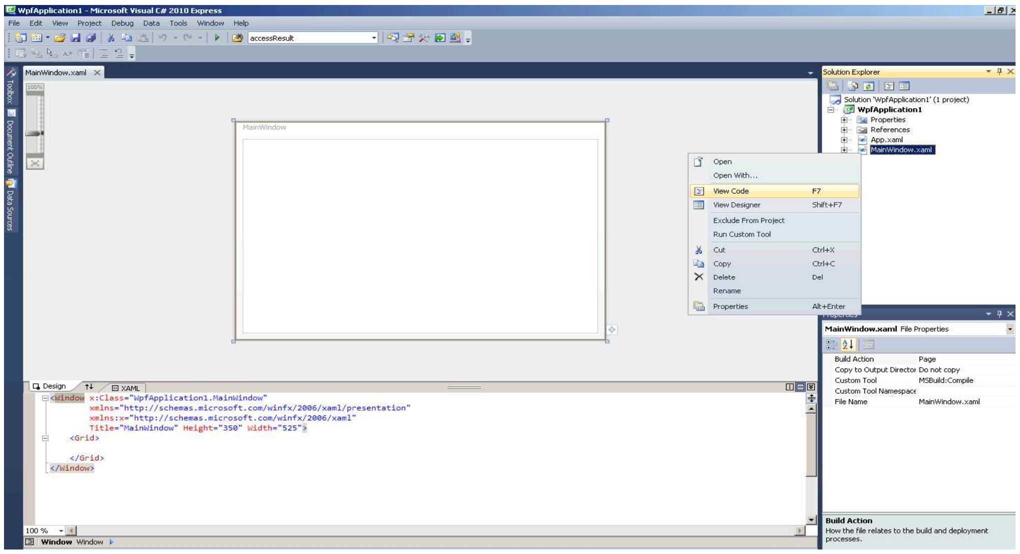

New Project
Recent Templates
Installed Templates
Visual C#
Online Templates
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
WPF Browser Application Visual C#
Empty Project Visual C#
Type: Visual C#
Windows Presentation Foundation client application
OK
Cancel

Name: WpfExample

Figure 10 New WFP Project Dialog

2. Right click on *MainWindow.xaml* and then select “View Code”.

3. Enter your code for the WPF example which runs behind the form. See Figure 12.



```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Figure 11 WPF Code form

4. See Figure 13 below which is a sample form and the associated XAML text.

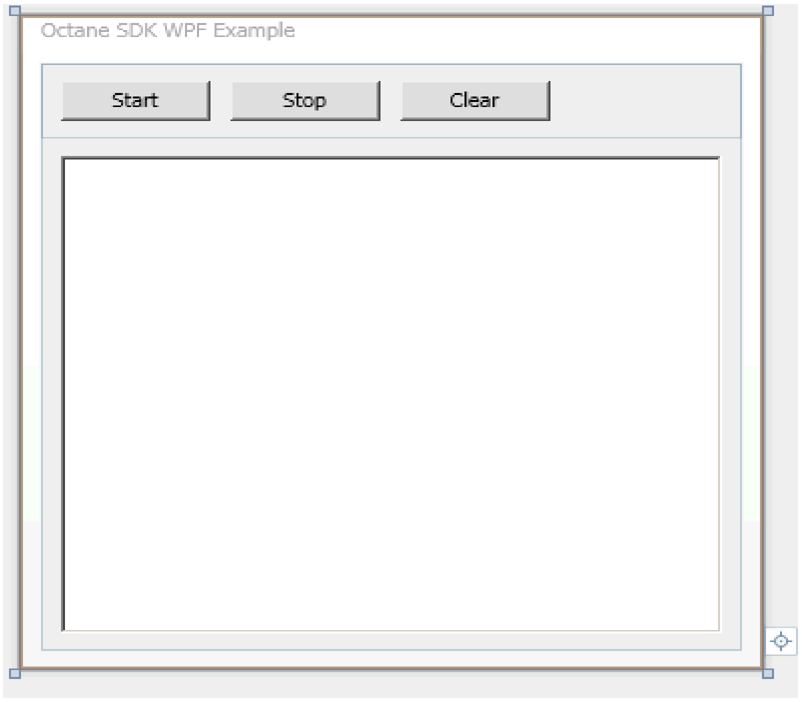


Figure 12 WPF Form Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Impinj.OctaneSdk;

namespace WpfExample
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        // Create an instance of the SpeedwayReader class.
        private SpeedwayReader Reader = new SpeedwayReader();

        // Declare a delegate to handle tag reports.
        // The TagsReported event handler cannot operate directly
        // on UI elements becuase it runs in a separate thread.
        private delegate void TagsReportedDelegate(List<Tag> tag);

        public MainWindow()
        {
            InitializeComponent();
            try
            {
                // Connect to the reader.
                // Replace "SpeedwayR-xx-xx-xx" with your
                // reader's host name or IP address.
                Reader.Connect("SpeedwayR-xx-xx-xx");

                // Remove all settings from the reader.
                Reader.ClearSettings();

                // Get the factory default settings
                // We'll use these as a starting point
                // and then modify the settings we're
                // interested in
                Settings settings = Reader.QueryFactorySettings();

                // Tell the reader to include the antenna number
                // in all tag reports. Other fields can be added
                // to the reports in the same way by setting the
                // appropriate Report.IncludeXXXXXXX property.
                settings.Report.IncludeAntennaPortNumber = true;

                // Send a tag report for every tag read.
                settings.Report.Mode = ReportMode.Individual;

                // Apply the newly modified settings.
                Reader.ApplySettings(settings);

                // Assign the TagsReported handler.
                // This specifies which function to call
                // when tags reports are available.
                // This function will in turn call a delegate
                // to update the UI (Listbox).
                Reader.TagsReported += new EventHandler
                    <TagsReportedEventArgs>(OnTagsReported);
            }
            catch (OctaneSdkException ex)
            {
                // An Octane SDK exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An Octane SDK exception has occured : {0}",
                              ex.Message);
            }
            catch (Exception ex)
            {
                // A general exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An exception has occured : {0}", ex.Message);
            }
        }

        private void buttonStart_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                // Don't call the Start method if the
                // reader is already running.
                if (!Reader.QueryStatus().IsSingulating)
                {
                    // Start reading.
                    Reader.Start();
                }
            }
            catch (OctaneSdkException ex)
            {
                // An Octane SDK exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An Octane SDK exception has occured : {0}",
                               ex.Message);
            }
            catch (Exception ex)
            {
                // A general exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An exception has occured : {0}", ex.Message);
            }
        }

        private void updateListbox(List<Tag> list)
        {
            // Loop through each tag is the list and add it to the Listbox.
            foreach (var tag in list)
            {
                listTags.Items.Add(tag.Epc + ", " + tag.AntennaPortNumber);
            }
        }

        private void OnTagsReported(object sender, TagsReportedEventArgs args)
        {
            // This function gets called when a tag report is available.
            // Since it is executed in a different thread, we cannot operate
            // directly on UI elements (the Listbox) in this function.
            // We must execute another function (updateListbox) on the main thread
            // using BeginInvoke. We will pass updateListbox a List of tags.
            TagsReportedDelegate del = new TagsReportedDelegate(updateListbox);
            this.Dispatcher.BeginInvoke(del, args.TagReport.Tags);
        }

        private void buttonStop_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                // Don't call the Stop method if the
                // reader is already stopped.
                if (Reader.QueryStatus().IsSingulating)
                {
                    Reader.Stop();
                }
            }
            catch (OctaneSdkException ex)
            {
                // An Octane SDK exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An Octane SDK exception has occured : {0}",
                               ex.Message);
            }
            catch (Exception ex)
            {
                // A general exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An exception has occured : {0}", ex.Message);
            }
        }

        private void buttonClear_Click(object sender, RoutedEventArgs e)
        {
            // Clear all the readings from the Listbox.
            listTags.Items.Clear();
        }

        private void Window_Closing(object sender,
                                      System.ComponentModel.CancelEventArgs e)
        {
            // The application is closing.
            // Stop the reader and disconnect.
            try
            {
                // Don't call the Stop method if the
                // reader is already stopped.
                if (Reader.QueryStatus().IsSingulating)
                {
                    Reader.Stop();
                }
                // Disconnect from the reader.
                Reader.Disconnect();
            }
            catch (OctaneSdkException ex)
            {
                // An Octane SDK exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An Octane SDK exception has occured : {0}",
                               ex.Message);
            }
            catch (Exception ex)
            {
                // A general exception occurred. Handle it here.
                System.Diagnostics.Trace.
                    WriteLine("An exception has occured : {0}", ex.Message);
            }
        }
    }
}
```

Notices:

www.impinj.com

Powered By Impinj ®

The "Powered by Impinj" shield is your assurance of RFID integrity