SensArray Programming Guide

Legal Notices

FCC NOTICE: This kit is designed to allow:

(1) Product developers to evaluate electronic components, circuitry, or software associated with the kit to determine whether to incorporate such items in a finished product and

(2) Software developers to write software applications for use with the end product. This kit is not a finished product and when assembled may not be resold or otherwise marketed unless all required FCC equipment authorizations are first obtained. Operation is subject to the condition that this product not cause harmful interference to licensed radio stations and that this product accept harmful interference. Unless the assembled kit is designed to operate under part 15, part 18 or part 95 of this chapter, the operator of the kit must operate under the authority of an FCC license holder or must secure an experimental authorization under part 5 of this chapter.

DCN-ST-00001-D   15 November 2021

Version 1.5 Revision History

Version	Author	Date	Changes
0.1	Brad Gaiser	2017-10-18	First Draft
1.0	Brad Gaiser	2017-11-17	Updated to reflect various name changes. Cleanup for initial customer release.
1.1	Brad Gaiser	2018-01-29	• Documented the callback methods and usage for reporting tag epc data during continuous inventory cycles that are started and stopped by the methods StartContinuousRead() and StopContinuousRead. The changes to support this are in a new overload for the constructor and an interface for setting the callback explicitly as well as a more detailed description of how to use the callback under the StartContinuousRead() method. • Added methods to allow the reader to read all the tag data banks, write to them, lock them and to kill tags. These are supported by the new methods ReadTagData(), WriteTagData(), LockTagData(), and KillTag().
1.2	Brad Gaiser	2018-02-12	• Added helper method SetSessionParameter • Added methods for getting and setting the Search Mode (GetSearchMode, SetSearchMode) • Added methods for getting reader name, type, and configuration (GetReaderName, GetReaderType, GetReaderConfig).
1.3	Brad Gaiser	2018-03-27	• Added methods for setting and clearing inventory filters: SetInventoryFilter() and ClearInventoryFilter() • Added methods for getting the region setting of the reader: GetRegionSetting() • Provided additional comments to clarify the functionality of other methods.
1.4	Brad Gaiser	2018-05-10	• Added method for querying which antennas are connected: GetConnectedAntennas
1.5	Robert Ma	2019-4-19	• Corrected errors in text in Command descriptions
1.5c	Robert Ma	2019-6-14	• Added new commands to Appendix B • Added new Appendix D • Added new Appendix E: Unsupported Commands
1.6	Robert Ma	2019-6-16	• Added Bluetooth and Wi-Fi- Configuration Commands
1.7	Neil Mitchell	2021-11-15	• Copyright updates only

Contents

Introduction

This guide is intended to help a first-time programmer writing code to control the SensArray get over the initial hurdle of communicating with the device. As such, we will present a couple of simple console applications that communicate with the reader to obtain data and display it in a console window. We are using a console application as the basis for our sample code because it keeps the examples simpler than would be the case if we tried to illustrate the concepts in the context of a graphical user interface.

The first two examples below utilize the Application Program Interface (API) layer code to do a couple of standard tasks. The first sends a query to the reader to have it return its current firmware version. From that foundation, we expect that you will be able to send queries, use the API to set reader and module parameters, and reread those value to confirm that they were set properly.

The second example utilizing the API illustrates how to typically start and stop a tag inventory read cycle. This example is more complex because once a read cycle is started, the reader starts sending a continuous stream of data to your application and will only respond to the stop command. Because of these constraints, the cleanest way to architect this type of application is to use a separate thread for reading the tags while the main thread sends the stop command based on an event. In this example, we will use a timer to read tags for 10 seconds then send the stop command. This, of course, is not the only type of event that could be used to stop the read cycle (a key press could be used, or a button click in a graphical user interface could be used, etc.). We chose a timer for this example due to its simplicity.

We have also provided a third example which uses the low-level communications API to send the raw byte array commands to the reader and to receive the corresponding responses. We will provide the application that is equivalent to the first example where we request the SensArray's firmware ID and display its value in a console window.

What We Won't Cover Here

The intended audience for this document is the programmer who needs to get a quick start learning how to use the command-encapsulation API and/or low-level communications API before writing his or her RFID application. This document is not intended as a tutorial for a programmer that needs to understand all the ins and outs of how to develop a reliable, robust application for their end application.

This doesn't cover programming at the binary protocol level. SensThys offers two additional documents for binary programming. The first details the higher-level reader commands that are not handed off to the reader module inside the reader: _SensArray Communications Protocol, Part I._ The second documents the binary protocol for communicating directly with the internal module. This includes commands for setting and getting the read and write power levels, setting the reader protocols, performing tag reads and writes, and similar commands. This document is entitled: _SensArray Communications Protocol, Part II_. Both documents can be downloaded from the SensThys.com website.

DCN-ST-00001-D   15 November 2021

What Else Is Here?

At the end of this document, you will find five Appendixes:

- Appendix A provides the code for each of the example in text form so that it is easy to copy and paste the code from this document into the Visual Studio code editor rather than type it in.
- Appendix B provides detailed documentation of the API that encapsulates (hides) the binary commands behind more readable function calls.
- Appendix C documents the lower-level communication layer that does the work of setting up the Ethernet communication socket, packages up the binary data streams, sends the message to the reader, and finally receives the replies to those messages.
- Appendix D documents a utility class provided by the API for listening for and providing callbacks for messages that a reader sends out when various reader events occur.
- Appendix E discusses what the API does when a method is called that is not supported on a specific hardware platform.

Language Support

The versions of the command-encapsulation API and the low-level communications API that are documented here were written using Microsoft Windows .Net version 3.5. Consequently, for this version, we support C# and VB.Net. Developing programs in each of these languages is reasonably straightforward using Visual Studio 2017 as the Integrated Development Environment (IDE). We understand that you may well have older version of Visual Studio or other IDEs that you might use for code development, however, in the interest of getting the first version written and available, we will restrict our discussion and illustrations to Visual Studio 2017.

One additional comment is in order. We are moving toward providing a broader development capability than what is currently documented here. We understand that people may want to develop their applications in other languages – C++, C, Java, Python, and so on. To support that broader effort, we will first provide COM-enabled versions of these APIs, but will also provide documentation of basic requirements for communicating with the device at the socket programming level.

Getting Started

To get started, you need to do a couple of things. The first is to install the libraries (dll's) that are discussed in this document. The second is to configure and gain some basic familiarity with operating the reader.

Both objectives are best achieved by reading through the SensArray User's Guide. In that guide, you will find the instructions for installing the SensArray graphical user interface (GUI). The APIs that are discussed in this document are installed as part of the installation of

the GUI. The User's Guide also discusses how to quickly learn to use the GUI to read tags and work with it to perform some basic configuration and deployment tasks.

Once you have obtained a basic familiarity with the reader, you are ready to move on . . .

Example 1: Hello World in C# Land

In this section, we will create a simple program in C# under Visual Studio 2017. As always, we will start with the simplest program that does something useful. In this case, we will set up the reader's communication information, instantiate the class for communicating with the reader, and request and print the reader's firmware revision number.

Creating the new Visual Studio C# project

1. Start Visual Studio
2. Create a new project (File/New/Project...)
3. Select "Window Classic Desktop" under Visual C# on left
4. Select "Console App" in center pane
5. Name the project (RFID_HelloWorld) and specify where you want it to reside
6. Hit OK
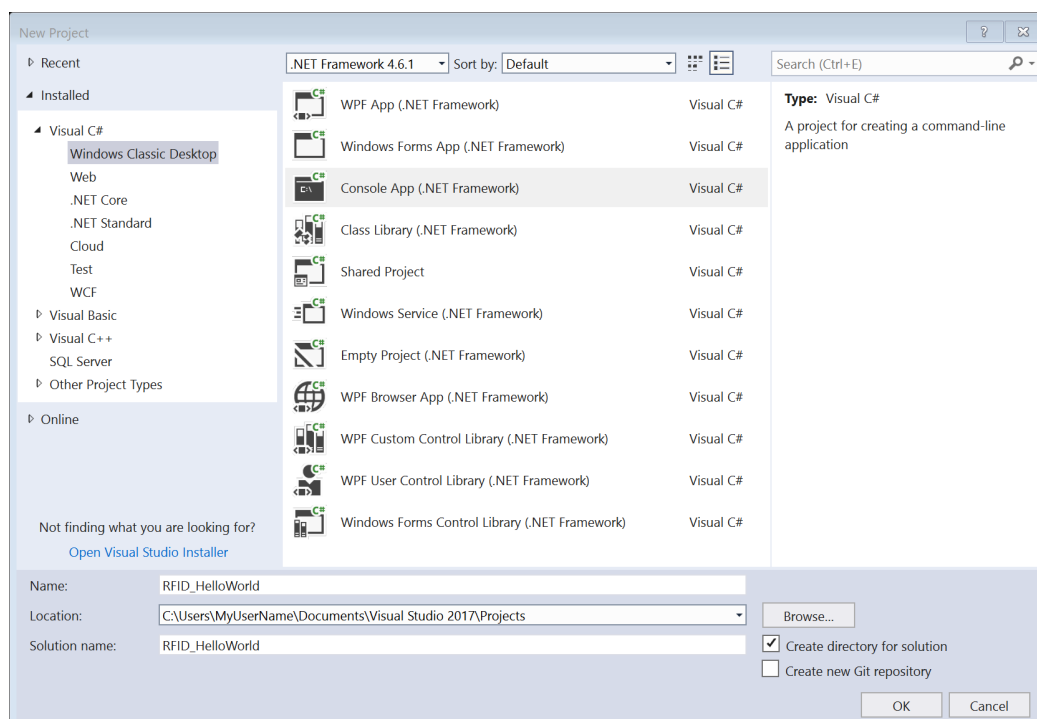
Setting up the library reference needed to access the RFID.Reader namespace

1. Open the Solution Explorer
2. Expand the RFID_HelloWorld line, if needed
3. Right click on the References line and click the "Add Reference . . ." menu item
4. Browse for RFID.dll
   If this was installed in its standard location, you will find it under "C:\Program Files (x86)\SensThys\RFID Console". If you change to this folder, then search for RFID.dll, you should find the correct path to this library. Select that path as the path for the Reference.
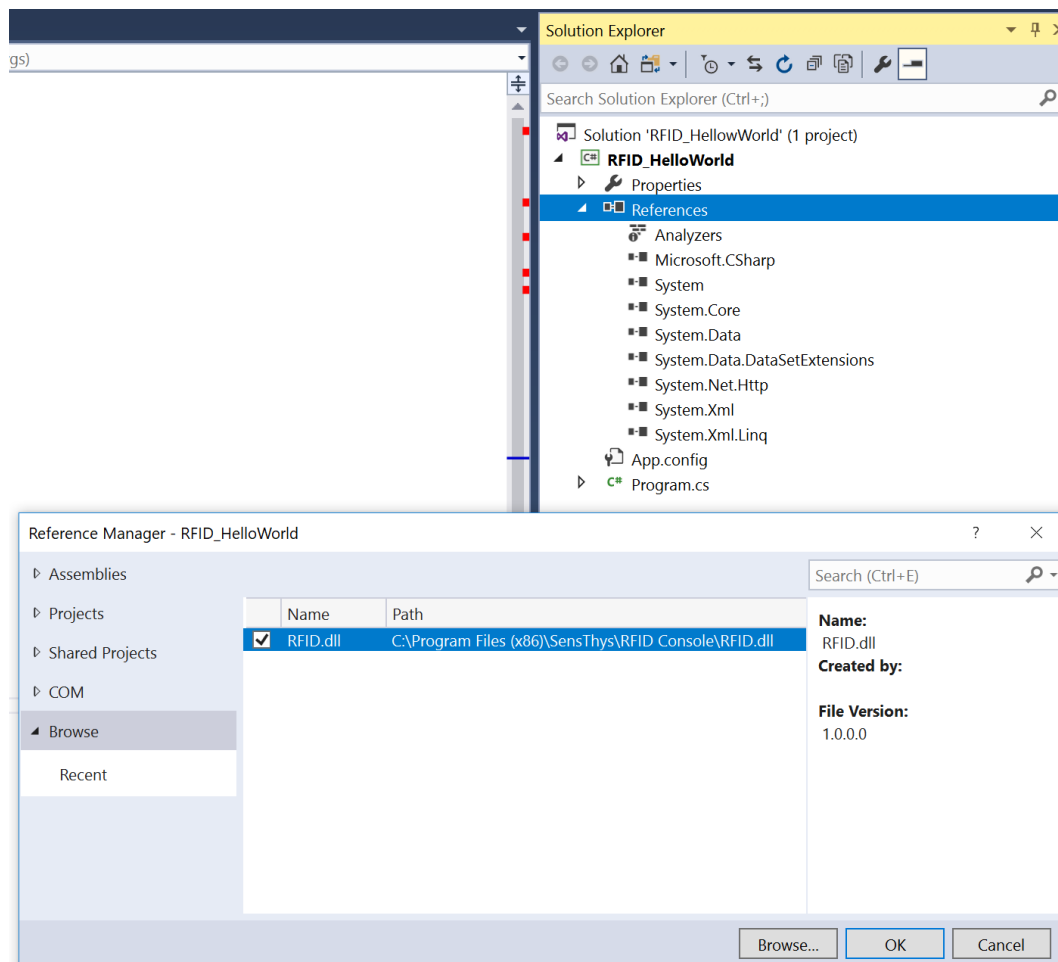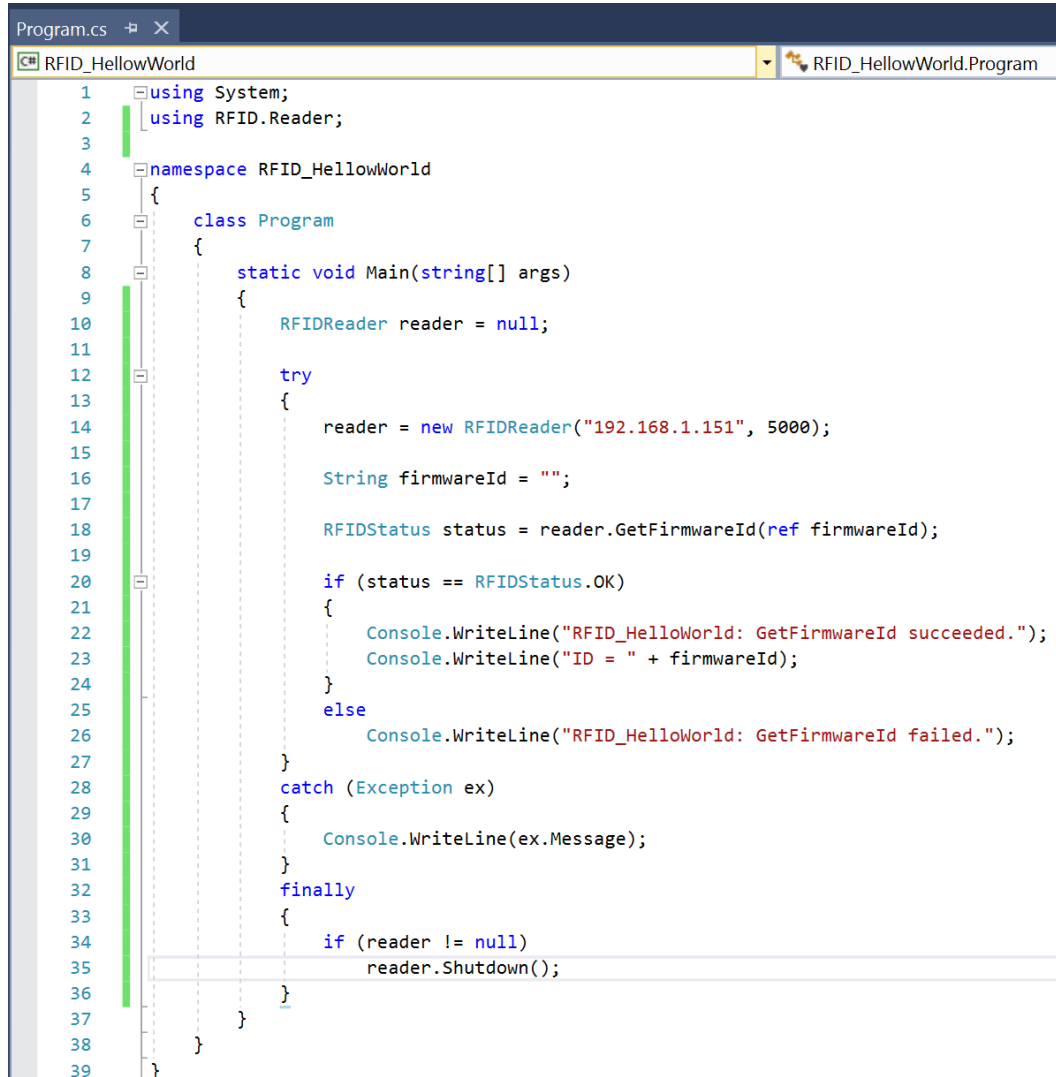
The Code

Again, you should find the file Program.cs under the Solution Explorer for the new project you just created. If Program.cs is not yet open in an editor window, double click on it to open it for editing.

Type in (or copy and paste from Appendix A) the following code:

```csharp
using System;
using RFID.Reader;

namespace RFID_HellowWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            RFIDReader reader = null;

            try
            {
                reader = new RFIDReader("192.168.1.151", 5000);

                String firmwareId = "";

                RFIDStatus status = reader.GetFirmwareId(ref firmwareId);

                if (status == RFIDStatus.OK)
                {
                    Console.WriteLine("RFID_HelloWorld: GetFirmwareId succeeded.");
                    Console.WriteLine("ID = " + firmwareId);
                }
                else
                    Console.WriteLine("RFID_HelloWorld: GetFirmwareId failed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (reader != null)
                    reader.Shutdown();
            }
        }
    }
}
```

The following are the lines you need to add to the file that was created when you selected the Console App option:

1. First, add "using RFID.Reader;" as seen at line 2. This simplifies the references to the RFIDReader class and its methods by specifying you want to use the RFID.Reader namespace. Note that when you initially create this project, the file Program.cs will

include several using statements at the very top of the file that are not needed to run this example. You can leave those if you want or delete all except the using System statement as shown here.

2. Second, you will be adding lines 10 through 36. This implements the basic functionality of this simple application. The purpose of the various parts of this program are described here:

   a. A quick look at lines 12 through 36 shows that the entire process is wrapped in a try/catch/finally statement. In setting up the RFIDReader class and communicating with the reader, several exceptions can be thrown. Consequently, robust error management when communicating with the reader requires that your code captures and handles unexpected problems cleanly.

   b. At line 14, we create a new instance of the RFIDReader class passing in the IP address of the reader that you want to query and its port number. In this example, the IP address assigned to the controller was 192.168.1.151, and the TCP port number was not changed from its default value of 5000. If you don't know the IP address of the reader you are working with, the easiest way to determine it is to open the RFID Console application and see what IP address is being shown in the Reader List window.

   c. At line 16, we allocate a string variable to hold the firmware id to be returned by the reader.

   d. At line 18, we call the GetFirmwareId method, passing the firmwareId variable by reference. Note that all the methods in the RFIDReader that communicate with the reader have their returned results passed back through the parameter list, hence the call by reference. If for some reason, the reader finds one or more of the parameter values to be out of range, that error will be indicated by the method call returning RFIDStatus.FAILED. If the parameters are error free, the return value will be RFIDStatus.OK.

   e. In lines 20 through 36, we handle the returned results and any errors that might occur. In this case, the else clause at lines 25 and 26 is not needed because there are no parameters that might be out of range. We wanted to illustrate how the status return value might be handled if there was a possible data error.

   f. At lines 28-31, we have provided a generic exception handler for the handful of exceptions that might be raised. The first that might occur is in the instantiation of the RFIDReader class. In checking to be sure that the IP address is correct, a System.FormatException would be raised if the IP address was not in the standard 4-field dotted notation typically used to specify IP addresses. Secondly, if the wrong IP address is specified, or there is some communication problem with the reader, a System.TimeoutException will be raised. Other exceptions related to opening TCP sockets might be raised as well, so it is generally good

practice to set up an exception handler that can allow your program to continue properly if something unexpected should arise.

g. Finally, lines 32-35 show how to provide final cleanup of the underlying communication channel. The Shutdown() method closes the socket and disposes of any memory or other resources that the socket might hold.

Compiling and Testing Your Program

Now that you have created the code, you need to compile and test it. Under Visual Studio 2017, you do this using the following procedure:

1. Compile the code using the menu item "Build/Build Solution". This should open an Output window in the Visual Studio development environment showing the compiler progress. If the compiler detects any errors in your code, it will open an Error List window to show you those errors.
2. Correct any errors and recompile using "Build/Build Solution" until your code is error free.
3. Finally, test your code by using the menu item "Debug/Start Without Debugging" or by hitting Ctrl+F5. This will open a console window where the printed output from your program will be displayed and paused waiting for you to "Press any key to continue . . .". Once you hit a key, the window will be closed. If you have entered your IP address and port number correctly, you should see the output shown here:

It is possible that you might see a newer firmware version number, but it should have the format (V#.#.#) shown.

One note: If you use "Debug/Start Debugging" rather than "Debug/Start Without Debugging", the console window will open up and then close without your being able to see the printed content.

Example 2: Reading Tags in Multi-Threaded C# Land

In this section, we will create a program in C# under Visual Studio 2017 that will read tags from the reader. The main problem with reading tags is that once tag reads are started, issuing the command to stop the read cycle typically would need to happen asynchronously to the loop where the tags are being read. In a GUI-based application, the tag reads would happen in a tight loop in a thread outside the main window event handling loop. That way, the reads can happen as fast as possible without blocking the GUI from handling user input or displaying content.

In this example, we reverse that process and create a timer which operates in a separate process so that the main thread can read tags as quickly as possible, but when the timer elapses, it is able to issue the command to stop the tag reads. Due to networking delays, tag data may still be queued up, so the read loop continues until the method for reading tag data indicates that the stop command has been received and the last of the tag data has been sent.

Creating the new Visual Studio C# project

1. Start Visual Studio
2. Create a new project (File/New/Project...)
3. Select "Window Classic Desktop" under Visual C# on left
4. Select "Console App" in center pane
5. Name the project (RFID_ReadTags) and specify where you want it to reside
6. Hit OK

Setting up the library reference needed to access the RFID.Reader namespace

1. Open the Solution Explorer
2. Expand the RFID_ReadTags line, if needed
3. Right click on the References line and click the "Add Reference..." menu item
4. Browse for RFID.dll
   If this was installed in its standard location, you will find it under "C:\Program Files (x86)\SensThys\RFID Console". If you change to this folder, then search for RFID.dll, you should find the correct path to this library. Select that path as the path for the Reference.

The Code

You should find the file Program.cs under the Solution Explorer for the new project you just created. If Program.cs is not yet open in an editor window, double click on it to open it for editing.

Type in the code shown below or copy and paste the code from Example 2 in Appendix A. You will need to add the using statement at line 3, the declarations at lines 9-11, the application code at lines 15-78 in Main(), and the callback method StopOnTimerEvent() at lines 81-85.

***There is one thing that you need to note*: If for some reason your program should fail and exit while in the middle of the read loop, there may be cases where the stop command is not issued. If that should happen, you will not see the "Continuous read exiting normally" message and you should probably power cycle your reader to be sure that the module is reset.**

If the stop command is not received by the reader, the module will continue reading tags indefinitely. If your duty cycle is set too high, the RFID module may start to run excessively hot, shortening its life.

This example program shows how defensive programming can be applied to handle this condition. Here, the finally clause of the try/catch block is used to ensure that the timer can fire off the stop command even if an exception is thrown. However, if you were to kill this program before it can run to completion, the module could still be left in a running state.

Program.cs

RFID_ReadTags   RFID_ReadTags.Program   Main(string[] args)

```csharp
1   using System;
2   using System.Threading;
3   using RFID.Reader;
4
5   namespace RFID_ReadTags
6   {
7       class Program
8       {
9           static RFIDReader reader = null;
10          static Timer tenSecondTimer = null;
11          static Boolean stopCommandIssued;
12
13          static void Main(string[] args)
14          {
15              try
16              {
17                  String firmwareId = "";
18                  String epc = "";
19                  Double rssi = 0.0;
20                  Byte antennaNumber = 0;
21                  Int32 tagnum = 0;
22
23                  // Create/initialize the controller
24                  reader = new RFIDReader("192.168.1.151", 5000);
25
26                  RFIDStatus status = reader.GetFirmwareId(ref firmwareId);
27
28                  if (status == RFIDStatus.OK)
29                      Console.WriteLine("Reading tags for reader with firmware id: " + firmwareId);
30                  else
31                  {
32                      Console.WriteLine("Unable to start the program. Failed to get firmware id from reader.");
33                      return;
34                  }
35
36                  // Create the timer
37                  Timer tenSecondTimer = new Timer(StopOnTimerEvent, reader, 10000, 10000);
38
39                  // Issue the command to start reading tags.
40                  status = reader.StartContinuousRead();
41
42                  // Read tags until status == CommandStatus.DONE
43                  while (status != RFIDStatus.DONE)
44                  {
45                      tagnum++;
46                      status = reader.ReadNextTag(ref epc, ref rssi, ref antennaNumber);
47                      switch (status)
48                      {
49                          case RFIDStatus.OK:
50                              Console.WriteLine("Tag #:" + tagnum.ToString() + ", Ant #: " +
51                              antennaNumber.ToString() + ", EPC: " + epc);
52                              break;
53                          case RFIDStatus.FAILED:
54                              Console.WriteLine("Tag #:" + tagnum.ToString() + " encountered read error.");
55                              break;
56                          case RFIDStatus.DONE:
57                              Console.WriteLine("Continuous read exiting normally.");
58                              break;
59                      }
60                  }
61              }
62              catch (Exception ex)
63              {
64                  Console.WriteLine(ex.Message);
65              }
66              finally
67              {
68                  do
69                  {
70                      System.Threading.Thread.Sleep(100);
71                  } while (tenSecondTimer != null && !stopCommandIssued);
72
73                  if (tenSecondTimer != null)
74                      tenSecondTimer.Dispose();
75
76                  if (reader != null)
77                      reader.Shutdown();
78              }
79          }
80
81          static public void StopOnTimerEvent(Object reader)
82          {
83              ((RFIDReader)reader).StopContinuousRead();
84              stopCommandIssued = true;
85          }
86      }
87  }
88
```

73 %

The following describes the various sections of code needed to implement this application. We will not go into detail regarding the exception handling section or the use of the status return values except for where it is used to stop the tag read loop. A description of the exception handling and status return handling is described in Examples 1 and 3 in detail.

The following describe the code that you will be adding:

1. At lines 2 and 3, you need to specify the System.Threading namespace for the Timer that will be created and used to stop the continuous read cycle and the RFID.RFIDReader namespace for the RFIDReader class and the RFIDStatus return type. The RFID.RFIDReader namespace is describe in more detail in Example 1 above.

2. In lines 15 through 32, we create an instance of the RFIDReader class and verify connectivity by reading and printing the firmware id as we did in Example 1.

3. On line 35, we send the command to the controller to start a continuous read cycle by calling the StartContinuousRead() method.

4. After that, at line 37, we create a timer that will fire off for 10 seconds allowing for 10 seconds of continuous tag reads. Note that the callback that is invoked when the timer expires is defined at lines 81 through 85. The only thing the callback does is to issue the stop command to the reader by calling the StopContinuousRead() method for the controller object created in the main program.

5. Lines 43 through 60 define the main loop for receiving and printing the tag data from the reader. The tag read data stream is started by a successful call to StartContinuousRead(). If that call were to fail, the data loop would not be entered. Note that ReadNextTag(), which is the core method for receiving the tag data returns RFIDStatus.OK when a tag has been read, and returns RFIDStatus.DONE when the StopContinuousRead() command has been processed by the reader and the last of the queued tag data has been received. StopContinuousRead() stops the read cycle immediately, but there may be tag data that is queued up. The response to the stop command is queued up after the data for the last tag read, so the read loop can be exited once RFIDStatus.DONE is finally received. Note that under some circumstances, ReadNextTag() may encounter a bad read. In that case, it will return a status of RFIDStatus.FAILED. Lines 47 through 58 show a simple way to handle all the various return values from the ReadNextTag().

6. The exception handling in this case is very simple, however, as described above, we use the finally clause to wait for the timer to fire and send the stop command. Note that we created the timer just prior to sending the StartContinuousRead() command so that it would be null if an exception was thrown prior to the start read being issued. In that case, the do/while loop will exit after sleeping for 100 ms and not get stuck waiting for the stopCommandIssued indicator to change. The finally clause is also responsible for final cleanup of the timer object and disposing of the controller object.

DCN-ST-00001-D   15 November 2021

Compiling and Testing Your Program

Now that you have created the code, you need to compile and test it. Under Visual Studio 2017, you do this using the following procedure:

1. Compile the code using the menu item "Build/Build Solution". This should open an Output window in the Visual Studio development environment showing the compiler progress. If the compiler detects any errors in your code, it will open an Error List window to show you those errors.

2. Correct any errors and recompile using "Build/Build Solution" until your code is error free.

3. Finally, test your code by using the menu item "Debug/Start Without Debugging" or by hitting Ctrl+F5. This will open a console window where the printed output from your program will be displayed and paused waiting for you to "Press any key to continue . . .". Once you hit a key, the window will be closed. If you have entered your IP address and port number correctly, you should see output similar to what is shown here:

```
C:\WINDOWS\system32\cmd.exe

Reading tags for reader with firmware id: V1.0.5
Tag #:1, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:2, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:3, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:4, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:5, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:6, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:7, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:8, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:9, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:10, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:11, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:12, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:13, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:14, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:15, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:16, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:17, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:18, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:19, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:20, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:21, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:22, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:23, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:24, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:25, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:26, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:27, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:28, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:29, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:30, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:31, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:32, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:33, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:34, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:35, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:36, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:37, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:38, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Tag #:39, Ant #: 0, EPC: B9-F5-3C-0A-B3-10-F1-16-03-92-8D-25
Tag #:40, Ant #: 0, EPC: 13-E6-EC-E6-2D-25-3E-47-EA-A8-EE-60
Continuous read exiting normally.
Press any key to continue . . .
```

7. It is possible that you might see a newer firmware version number, but it should have the format (V#.#.#) shown.

    One note: If you use "Debug/Start Debugging", rather than "Debug/Start Without Debugging", the console window will open up, then close without your being able to see the printed content.

Example 3: Hello World in Binary Protocol Land

In this section, we will create a simple program in C# under Visual Studio 2017 that is completely equivalent to the example in Example 1 above where we request and print the reader's firmware ID except that we write the code using the low-level reader binary protocol.

Creating the new Visual Studio C# project

1. Start Visual Studio

2. Create a new project (File/New/Project . . .)
3. Select Window Classic Desktop under Visual C# on left
4. Select Console App in center pane
5. Name the project (RFID_HelloWorld2) and specify where you want it to reside
6. Hit OK

Setting up the library reference needed to access the RFID.ReaderComm namespace
1. Open the Solution Explorer
2. Expand the RFID_HelloWorld2 line, if needed
3. Right click on the References line and click the "Add Reference . . ." menu item
4. Browse for RFID.dll
   If this was installed in its standard location, you will find it under "C:\Program Files (x86)\SensThys\RFID Console". If you change to this folder, then search for RFID.dll, you should find the correct path to this library. Select that path as the path for the Reference.

The Code

You should find the file Program.cs under the Solution Explorer for the new project you just created. If Program.cs is not yet open in an editor window, double click on it to open it for editing.

Type in the code highlighted by the green bars below. You will need to add line 2 and lines 10-46.

```csharp
Program.cs  ⇄ X
RFID_HelloWorld2                                          RFID_HelloWorld2.Program
1   using System;
2   using RFID.ReaderComm;
3
4   namespace RFID_HelloWorld2
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              RFIDReaderComm readerComm = null;
11
12              try
13              {
14                  readerComm = new RFIDReaderComm("192.168.1.151", 5000);
15
16                  Int32 MaxResponseLength = 10;
17                  Byte[] responseBuffer = new Byte[MaxResponseLength];
18
19                  const Byte GetReaderFirmwareIdCommand = 0x02;
20
21                  Int32 datalength = readerComm.SendMessage(ref responseBuffer,
22                                          MaxResponseLength,
23                                          ReaderSubsystem.RFID_Controller,
24                                          GetReaderFirmwareIdCommand);
25
26                  if (datalength == 3)
27                  {
28                      Console.WriteLine("RFID_HelloWorld2: Read Firmware ID succeeded.");
29                      Console.WriteLine(String.Format("ID = V{0}.{1}.{2}",
30                                          responseBuffer[0], responseBuffer[1], responseBuffer[2]));
31                  }
32                  else
33                  {
34                      Console.WriteLine("RFID_HelloWorld2: Read Firmware ID returned unexpected datalength.");
35                      Console.WriteLine("Expected 3 bytes, received " + datalength.ToString() + " bytes.");
36                  }
37              }
38              catch (Exception ex)
39              {
40                  Console.WriteLine(ex.Message);
41              }
42              finally
43              {
44                  if (readerComm != null)
45                      readerComm.Dispose();
46              }
47          }
48      }
49  }
50
88 %
```

You will need to add the following code to the file that was created when you selected the Console App option:

1. First, add "using RFID.ReaderComm;" as seen at line 2. This simplifies the references to the RFIDReaderComm class and its methods. Note that when you initially create this project, the file Program.cs will include several using statements at the very top of the file that are not needed to run this example. You can leave those if you want or delete all except the using System statement as shown here.
2. The code in lines 10 through 46 are as follows:

a. A quick look at lines 12 through 46 shows that the entire process is wrapped in a try/catch/finally statement. In setting up the connection and communicating with the reader, several exceptions can be thrown. Consequently, robust error management handling when communicating with the reader requires that your code captures and handles unexpected problems cleanly.

b. At line 14, we create a new instance of the RFIDReaderComm class passing in the IP address of the reader that you want to query and its port number. In this example, the IP address assigned to the controller was 192.168.1.151, and the TCP port number was not changed from its default value of 5000. As discussed in Example 1, if you don't know the IP address of the reader you are working with, the easiest way to determine it is to open the RFID Console application and see what IP address is being shown in the Reader List window.

c. At lines 16 and 17, we create a buffer to hold the output results.

d. At line 18, we define a constant for the command code used to retrieve the reader's firmware ID. Note that in this example, the command code is 2, and as will be seen in the next section, the command itself doesn't pass in any data. This is not typically the case, since many of the commands set various configuration parameters, in which case, data does need to be passed in. For commands that pass in data, you would pass in the data array and the number of bytes in the data as the 5th and 6th parameters to the SendMessage() call.

e. At lines 21-24, we call the SendMessage() method of the RFIDReaderComm class. The first two parameters (lines 21 and 22) pass in the buffer (responseBuffer) that will receive the actual data returned by the reader and the allocated size of that buffer (MaxResponseLength). Note that we only expect this command to return 3 bytes of data, but have allocated 10 bytes. If you were using a common buffer for all of the calls you need to make to the reader, you can over-allocate the required space as shown here.

f. On line 23, we pass in a parameter that specifies which of the internal reader's subsystems the command will be directed to. If you recall from the discussion in the section above "What Else Is Here?" there are two primary subsystems in the reader to which queries and commands can be directed. For this case, we are interested in the reader's firmware id (rather than the RFID module's firmware id), so we direct the query to the reader by specifying the ReaderSubsystem.RFID_Controller value for this parameter. (If we were querying the internal module's firmware ID, we would use the subsystem ReaderSubsystem.RFID_Module for this parameter.)

g. Finally, at line 24, we pass in the constant GetReaderFirmwareIdCommand as the command code for which we are requesting a response.
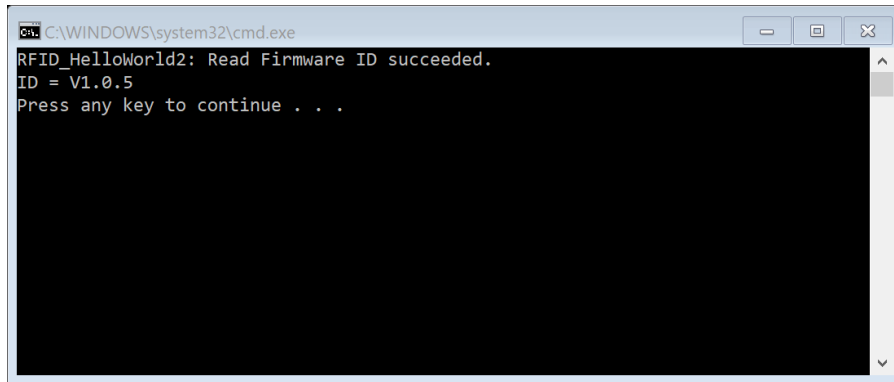
h. Note that there are two optional parameters to SendMessage() that would be used to pass input data to the reader as part of a command. In this case, the request for the reader's firmware id doesn't have any data content (other than the command code, which is handled through the 3 parameter) we don't include these two optional parameters.

i. Lines 26 through 46 handle the data returned from the reader or deal with any problems that might occur. Strictly speaking, the test and line 26 and the else clause in lines 32-36 are not needed since there should be no circumstances where an incorrect byte count will be returned without some other exception being raised. This was just included to help you understand the various returns from the call to SendMessage(). The catch clause that is part of the exception handling is shown at lines 38-41. There are a number of exceptions that might be raised. The first that might occur is in the instantiation of the RFIDReaderComm class. In checking to be sure that the IP address is correct, a System.FormatException would be raised if the IP address was not in the standard 4-field dotted notation typically used to specify IP addresses. Secondly, if the wrong IP address is specified, or there is some communication problem with the reader, a System.TimeoutException will be raised. Other exceptions related to opening TCP sockets might be raised as well, so it is generally good practice to set up an exception handler that can allow your program to continue properly if something unexpected should arise.

j. Lines 42-46 show the final cleanup needed for this class. Since the socket underlying the connection between your computer and the reader fall outside the memory handling in .Net, we implemented the Dispose() method to close the socket and cleanup any memory allocated when using the socket.

Compiling and Testing Your Program

Now that you have created the code, you need to compile and test it. Under Visual Studio 2017, you do this using the following procedure:

1. Compile the code using the menu item "Build/Build Solution". This should open an Output window in the Visual Studio development environment showing the compiler progress. If the compiler detects any errors in your code, it will open an Error List window to show you those errors.
2. Correct any errors and recompile using "Build/Build Solution" until your code is error free.
3. Finally, test your code by using the menu item "Debug/Start Without Debugging" or by hitting Ctrl+F5. This will open a console window where the printed output from your program will be displayed and paused waiting for you to "Press any key to continue . . .". Once you hit a key, the window will be closed. If you have entered your

IP address and port number correctly, you should see the output shown here:


```
C:\WINDOWS\system32\cmd.exe
RFID_HelloWorld2: Read Firmware ID succeeded.
ID = V1.0.5
Press any key to continue . . .
```

It is possible that you might see a newer firmware version number, but it should have the format (V#.#.#) shown.

One note: If you use "Debug/Start Debugging", rather than "Debug/Start Without Debugging", the console window will open up, then close without your being able to see the printed content.

Appendix A

This appendix will provide the code listings for the three examples in text form so that it is easier to copy and paste the code into the Visual Studio editors.

Example 1

```csharp
using System;
using RFID.Reader;

namespace RFID_HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            RFIDReader reader = null;

            try
            {
                reader = new RFIDReader("192.168.1.151", 5000);

                String firmwareId = "";

                RFIDStatus status = reader.GetFirmwareId(ref firmwareId);

                if (status == RFIDStatus.OK)
                {
                    Console.WriteLine("RFID_HelloWorld: GetFirmwareId succeeded.");
                    Console.WriteLine("ID = " + firmwareId);
                }
                else
                    Console.WriteLine("RFID_HelloWorld: GetFirmwareId failed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (reader != null)
                    reader.Shutdown();
            }
        }
    }
}
```

Example 2

```csharp
using System;
using System.Threading;
using RFID.Reader;

namespace RFID_ReadTags
{
    class Program
    {
        static RFIDReader reader = null;
        static Timer tenSecondTimer = null;
        static Boolean stopCommandIssued;

        static void Main(string[] args)
        {
            try
            {
                String firmwareId = "";
                String epc = "";
                Double rssi = 0.0;
                Byte antennaNumber = 0;
                Int32 tagnum = 0;

                // Create/initialize the controller
                reader = new RFIDReader("192.168.1.151", 5000);

                RFIDStatus status = reader.GetFirmwareId(ref firmwareId);

                if (status == RFIDStatus.OK)
                    Console.WriteLine("Reading tags for reader with firmware id: " + firmwareId);
                else
                {
                    Console.WriteLine("Unable to start the program. Failed to get firmware id from reader.");

                    return;
                }

                // Create the timer
                Timer tenSecondTimer = new Timer(StopOnTimerEvent, reader, 10000, 10000);

                // Issue the command to start reading tags.
                status = reader.StartContinuousRead();

                // Read tags until status == RFIDStatus.DONE
                while (status != RFIDStatus.DONE)
                {
                    tagnum++;
                    status = reader.ReadNextTag(ref epc, ref rssi, ref antennaNumber);
                    switch (status)
                    {
                        case RFIDStatus.OK:
                            Console.WriteLine("Tag #:" + tagnum.ToString() + ", Ant #: " +
                        antennaNumber.ToString() + ", EPC: " + epc);
                            break;
                        case RFIDStatus.FAILED:
                            Console.WriteLine("Tag #:" + tagnum.ToString() + " encountered read error.");
                            break;
                        case RFIDStatus.DONE:
                            Console.WriteLine("Continuous read exiting normally.");
                            break;
                    }
                }
```

```csharp
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            do
            {
                System.Threading.Thread.Sleep(100);
            } while (tenSecondTimer != null && !stopCommandIssued);

            if (tenSecondTimer != null)
                tenSecondTimer.Dispose();

            if (reader != null)
                reader.Shutdown();
        }
    }

    static public void StopOnTimerEvent(Object reader)
    {
        ((RFIDReader)reader).StopContinuousRead();
        stopCommandIssued = true;
    }
  }
}
```

Example 3

```csharp
using System;
using RFID.ReaderComm;

namespace RFID_HelloWorld2
{
    class Program
    {
        static void Main(string[] args)
        {
            RFIDReaderComm readerComm = null;

            try
            {
                readerComm = new RFIDReaderComm("192.168.1.151", 5000);

                Int32 MaxResponseLength = 10;
                Byte[] responseBuffer = new Byte[MaxResponseLength];

                const Byte GetReaderFirmwareIdCommand = 0x02;

                Int32 datalength = readerComm.SendMessage(ref responseBuffer,
                                                MaxResponseLength,
                                                ReaderSubsystem.RFID_Controller,
                                                GetReaderFirmwareIdCommand);

                if (datalength == 3)
                {
                    Console.WriteLine("RFID_HelloWorld2: Read Firmware ID succeeded.");
                    Console.WriteLine(String.Format("ID = V{0}.{1}.{2}",
                                    responseBuffer[0], responseBuffer[1], responseBuffer[2]));
                }
                else
                {
                    Console.WriteLine("RFID_HelloWorld2: Read Firmware ID returned unexpected
datalength.");
                    Console.WriteLine("Expected 3 bytes, received " + datalength.ToString() + " bytes.");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (readerComm != null)
                    readerComm.Dispose();
            }
        }
    }
}
```

Appendix B

API Interface Guide:
RFID.Reader Namespace

_____

public class RFIDReader

<u>Function:</u>

This class provides a high-level encapsulation of many of the commands that the sensor responds to for querying and changing the setup of the sensor as well as reading tags.

<u>Additional Information:</u>

A detailed description of how this class is used is described above with examples of how it is used. The general outline, though, is as follows:

- Instantiate an instance of the class passing in the IP address and the port number: RFIDReader sctrl = new RFIDReader("192.168.1.150", 5000);

- Call the method that you want to use: RFIDStatus = sctrl.GetModuleHardwareId(ref hwId);

- Process the result, then repeat for any additional commands that you wish to process.

- Close the connection when finished by calling sctrl.Shutdown();

Note that the underlying communication blocks with relatively long timeouts. Consequently, this is best used in a multi-threaded environment if these calls are made within the context of a graphical user interface.

One further comment on the various methods for communicating with the sensor. The command handler on the sensor does some value and range checking on the parameters that are being sent to it to change its configuration and to control its operation. Consequently, many of those commands return a success or failure value. We wanted the methods for communicating commands and receiving responses from the sensor to have a uniform structure. To accommodate achieving uniformity, returning a success or failure code, and communicating complex data we elected to indicate success or failure through the return value and to return data values from "Get" calls by reference through the parameter list.

The type returned from the query and update commands to the sensor is the enumeration RFIDStatus with the following three enumerators:

- FAILED: This value is returned when the command failed, typically because bad data was passed sent to the sensor.

- OK: This value is returned when the command succeeds.

- DONE: This value is returned when a sequence of calls needs to be made where the result is typically indicated by OK, but that the final response is indicated by returning DONE. Currently, the method GetNextTag() is the only interface that returns this value.

In addition to these return values, many of the methods will throw exceptions. Most of the methods in this API involve sending and receiving Ethernet messages. For a variety of reasons, there can be delays in the setup of the communication channel or delays in the actual communication process that result in timeout exceptions being the most common exception from these methods.

The design philosophy behind this division in the way errors are handled is to let the operating system and programming type of errors be handled through the exception handling process while data validation errors are handled through the return codes.

Communications setup and control

_____

constructor RFIDReader(String IPAddress, Uint16 PortNumber,
RFID.Reader.TagReadHandler continuousTagReadHandler)

Function:
This method creates an instance of the RFIDReader class and creates the underlying communications class while registering a callback that matches the TagReadHandler interface.

Parameters:

Parameter: IPAddress
Type - String
Description - Input parameter with dotted format IP address (e.g., "192.168.1.150").

Parameter: PortNumber
Type - UInt16
Description - Input parameter with the TCP port that the primary command communication will take place over.

Parameter: continuousTagReadHandler
Type - delegate void RFID.Reader.TagReadHandler(TagReadInfo tagInfo).

Where TagReadInfo is defined as:

public class TagReadInfo {
    public RFIDStatus status;
    public Byte antennaNumber;
    public String EPC;
    public Double RSSI;

    public TagReadInfo();

```
        public void Clear(); // Sets status=OK; other members to 0 or empty
        }
```

Description - This parameter specifies a callback routine that is called each time tag data is reported during a continuous inventory cycle.

Return Value:
An object instance of the RFIDReader class is returned when a new version of this class is instantiated.

Exception(s) thrown:
A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.

_____

constructor RFIDReader(String IPAddress, Uint16 PortNumber,
*   RFID.Reader.TagReadHandler2 continuousTagReadHandler)*

Function:
This method creates an instance of the RFIDReader class and creates the underlying communications class while registering a callback that matches the TagReadHandler2 interface.

Parameters:
    Parameter: IPAddress
        Type - String
        Description - Input parameter with dotted format IP address (e.g., "192.168.1.150").

    Parameter: PortNumber
        Type - UInt16
        Description - Input parameter with the TCP port that the primary command communication will take place over.

    Parameter: continuousTagReadHandler
        Type - delegate void RFID.Reader.TagReadHandler2(TagReadInfo tagInfo,
                                                        RFIDReader reader)

        See the definition of TagReadInfo shown above under the description for the constructor that accepts a callback of the TagReadHandler delegate type.

        Description - This parameter specifies a callback routine that is called each time tag data is reported during a continuous inventory cycle.

        This handler will pass back the RFIDReader object that the callback is registered with so that the application can determine the specific reader instance corresponding to the received tag data.

Return Value:

An object instance of the RFIDReader class is returned when a new version of this class is instantiated.

Exception(s) thrown:

A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.

_____

constructor RFIDReader(String IPAddress, Uint16 PortNumber)

Function:

This method creates an instance of the RFIDReader class and creates the underlying communications class. The parameters to this constructor are the same as documented above, but the callback is set to null. This constructor would be used if the application would like to call into GetNextTag directly rather than use the callback to have tag read info delivered to the application, if it would want to defer setting up the callback, or would be used simply use the reader instance to configure the reader, but will not be used to read tags.

Return Value:

An object instance of the RFIDReader class is returned when a new version of this class is instantiated.

Exception(s) thrown:

A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.

_____

void SetIPv4CommLink(String IPAddress, Uint16 PortNumber)

Function:

The IP address and port number are passed on to the underlying communication layer which closes the underlying socket if the IP address and port number don't match the values already set up.

Parameters:

    Parameter: IPAddress

        Type - String

        Description - Input parameter with dotted format IP address (e.g., "192.168.1.150").

    Parameter: PortNumber

        Type - UInt16

        Description - Input parameter with the TCP port that the primary command communication will take place over.

Return Value:
There is no return value from this method.

Exception(s) thrown:
A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.

Additional Information:
This method is used to reset the underlying socket whenever the sensor's network configuration has changed, or when the application chooses to communicate with a second sensor without wanting to create a separate instance of the RFIDReader class.

_____

SetTagReadCallback(RFID.Reader.TagReadHandler continuousTagReadHandler)

Function:
This method allows the continuous read callback function to be changed or disabled (by passing the parameter value as null).

Parameters:
    Parameter: continuousTagReadHandler
        Type - delegate void RFID.Reader.TagReadHandler(TagReadInfo tagInfo)
        The TagReadInfo structure is described under the RFIDReader constructor above.
        Description - This parameter specifies the callback routine that is called each time tag data is reported during a continuous inventory cycle.

Exception(s) thrown:
No exceptions will be thrown by this method.

_____

SetTagReadCallback(RFID.Reader.TagReadHandler2 continuousTagReadHandler)

Function:
This method allows the continuous read callback function to be changed or disabled (by passing the parameter value as null).

The callback that is registered with this overloaded method will pass a reference to the reader object with which it is registered back to the client code.

Parameters:
    Parameter: continuousTagReadHandler
        Type - delegate void RFID.Reader.TagReadHandler2(TagReadInfo tagInfo, RFIDReader reader)
        The TagReadInfo structure is described under the RFIDReader constructor above.

        Description - This parameter specifies the callback routine that is called each time tag data is reported during a continuous inventory cycle.

Exception(s) thrown:
No exceptions will be thrown by this method.

_____

void Shutdown()

Function:

This method closes down the underlying communications connection.

Return Value:

No value is returned from this method.

Exception(s) thrown:

System.Timeout as well as some socket exceptions may be thrown by this call. In general, these exceptions can be ignored.

Additional Information:

Once this method is called, that instance of the RFIDReader class will no longer be useable.

RF Module ID Queries

_____

RFIDStatus GetModuleHardwareId(ref String ModuleHwId)

Function:

Returns the RF Module's hardware identifier.

Parameters:

Parameter: ModuleHwId

Type - ref String

Description - The string representation of the module's hardware identifier is returned to the caller.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's hardware id. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The value returned has the format V#.#.# where the first field (#) is the major version level of the hardware, the second the minor version level, and the third field contains the revision number.

_____

RFIDStatus GetModuleFirmwareId(ref String ModuleFwNumber)

Function:

Returns the RF Module's firmware version number.

Parameters:

Parameter: ModuleFwNumber

Type - ref String

Description - The string representation of the module's firmware version number is returned to the caller.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's hardware id. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The value returned has the format V#.#.# where the first field (#) is the major version level of the firmware, the second the minor version level, and the third field contains the firmware's revision number.

_____

RFIDStatus GetModuleReaderId(ref String ReaderId)

Function:

This function returns the string value corresponding to the reader module's internal serial number.

Parameters:

Parameter: ReaderId

Type - ref String

Description - The 8-digit reader module's serial number is returned, formatted as a string.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's reader id. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

RF Setup and Query Methods

_____

RFIDStatus SetModulePowerSetting(Double ReadPower, Double WritePower)

Function:

This function directly sets the read and write power of the RFIDReader module.

Parameters:

Parameter: ReadPower

Type - Double

Description - The power to be used when reading tags in dBm. Valid range is from 5 to 30.

Parameter: WritePower
  Type - Double
  Description - The power to be used when writing tags in dBm. Valid range is from 5 to 30.

Return Value:
This function will return RFIDStatus.OK if it is able to set the RF module's read or write power level. It will return RFIDStatus.FAILED otherwise. This command will fail if ReadPower or WritePower is either less than 5 or greater than 30.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
Currently, the AntennaId should be set to 0.

_____

RFIDStatus SetAntennaPowerSetting(UInt32 AntennaId, Double ReadPower, Double WritePower)

Function:
Set the read and write power levels for when the reader is transmitting through the specified antenna.

Parameters:
  Parameter: AntennaId
    Type - UInt32
    Description - The number of the antenna for which the power is to be set. Range is from 0 to 3.

  Parameter: ReadPower
    Type - Double
    Description - The read power level in dBm to be set for the specified antenna when the reader will be reading tags. Must be in the range from 5 to 30 dBm.

  Parameter: WritePower
    Type - Double
    Description - The write power level in dBm to be set for the specified antenna when the reader will be write information to tags. Must be in the range from 5 to 30 dBm.

Return Value:
This function will return RFIDStatus.OK if it is able to set the read and write power level to be associated with the specified antenna. It will return RFIDStatus.FAILED otherwise. This command will fail if the AntennaId is not in the range from 0 to 3. It will also fail if either the ReadPower or the WritePower is less than 5 or greater than 30.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetModulePowerSetting(ref Double ReadPower, ref Double WritePower)

Function:

Returns the read power and write power levels from the reader module.

Parameters:

    Parameter: ReadPower

        Type - ref Double

        Description - Returned value of the current module's read power level in dBm.

    Parameter: WritePower

        Type - ref Double

        Description - Returned value of the current module's write power level in dBm.

Return Value:

This function will return RFIDStatus.OK if it is able to read the RF module's currently set read and write power levels. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

This command is complementary to the SetModulePowerSetting command. Note that if the power level has been changed by doing tag inventory reads, the power level is set to the values established by the SetAntennaPowerSetting commands.

_____

RFIDStatus GetPowerSetting(ref Double[] ReadPower, ref Double[] WritePower)

Function:

Returns the read power and write power level settings in dBm of the internal antenna (index 0) and the three external antennas (indexes 1, 2, and 3)

Parameters:

    Parameter: ReadPower

        Type - ref Double[]

        Description - Four element array returning the read power levels in dBm for each of the readers antennas.

    Parameter: WritePower

        Type - ref Double[]

        Description - Four element array returning the write power levels in dBm for each of the readers antennas.

Return Value:

DCN-ST-00001-D 15 November 2021

This function will return RFIDStatus.OK if it is able to read the read and write power levels for all of the antennas. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

An System.IndexOutOfRangeException will be thrown if the ReadPower and/or WritePower arrays do not have space for at least 4 values.

Additional Information:

This method requires that both arrays are at least four elements long even when interrogating a 2-port device. In that situation, the last 3 elements of each array are set to 0 dBm. Since the valid power range is always greater than 5 dBm, this acts as an indicator of the number of elements. In the future, the overloaded version with the number of antennas as an output can be used.

_____

RFIDStatus GetPowerSetting(out int NumberOfAntennas,
ref Double[] ReadPower,
ref Double[] WritePower)

Function:

Returns the read power and write power level settings in dBm of the internal antenna (index 0) and the three external antennas (indexes 1, 2, and 3)

Parameters:

    Parameter: NumberOfAntennas

        Type – out int

        Description - The number of antennas for the reader configuration (2-port vs. 3-port) is returned by this parameter. It will return 1 for the 2-port readers and 4 for the 3-port readers.

    Parameter: ReadPower

        Type - ref Double[]

        Description - Array at least "NumberOfAntennas" long returning the read power levels in dBm for each of the readers antennas.

    Parameter: WritePower

        Type - ref Double[]

        Description - Array at least "NumberOfAntennas" long returning the write power levels in dBm for each of the readers antennas.

Return Value:

This function will return RFIDStatus.OK if it is able to read the read and write power levels for all of the antennas. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

An System.IndexOutOfRangeException will be thrown if the ReadPower and/or WritePower arrays do not have space for at least "NumberOfAntennas" values.

_____

RFIDStatus GetGen2Params(ref Byte[] Gen2Params)

Function:

The Gen2Params data array is returned with the various Gen2 parameters filled in.

Parameters:

> Parameter: Gen2Params
>> Type - ref Byte[]
>> Description – This 8 byte array returns the Gen2 parameter settings for the RFIDReader module. Also, see the Additional Information below.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's Gen2 parameters. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

An System.IndexOutOfRangeException will be thrown if the Gen2Params array is not at least 8 bytes long.

Additional Information:

This method encapsulates the module command 0x22. See the "SensArray Communications Protocol, Part II" document for details on how the specific Gen2 parameters are encoded in Gen2Params data array.

_____

RFIDStatus SetGen2Params(Byte[] Gen2Params)

Function:

The Gen2Params for the RFIDReader module are set to the values specified in the Gen2Params data structure.

Parameters:

> Parameter: Gen2Params
>> Type – Byte[]
>> Description - This 8 byte array passes in the Gen2 parameter settings for the RFIDReader module.

Return Value:

This function will return RFIDStatus.OK if it is able to set the Gen2 parameters. It will return RFIDStatus.FAILED otherwise. This command will fail if an attempt is made to set values that are not valid for the specified data fields of the Gen2DataType structure.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

An System.IndexOutOfRangeException will be thrown if the Gen2Params array is not at least 8 bytes long.

Additional Information:

This method encapsulates the module command 0x20. See the "SensArray Communications Protocol, Part II" document for details on how the specific Gen2 parameters are encoded in Gen2Params data array.

_____

RFIDStatus SetSessionParameter(Byte SessionParam)

Function:

This method sets the Session Parameter.

Parameters:

    Parameter: SessionParam

        Type - Byte

        Description - This input parameter passes in the value for the session parameter to be set. The values are 0 through 3 for sessions 0 through 3.

Return Value:

This function will return RFIDStatus.OK if it is able to set the session parameter. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The session parameter is one component in the underlying byte protocol for setting the Gen2 parameters. This method reads the Gen2 parameters, changes the session bits, then sends the command to set the Gen2 parameters.

_____

RFIDStatus GetSearchMode(out Byte SearchMode)

Function:

Returns the current search mode for the reader module.

Parameters:

    Parameter: SearchMode

        Type - out Byte

        Description - The return value will be 0 if the dual search mode is active and 1 if single is active.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's search mode. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

RFIDReader.SearchModeDual and RFIDReader.SearchModeSingle are provided as symbolic constants to make code using this interface more readable.

_____

RFIDStatus SetSearchMode(Byte SearchMode)

Function:

Set the search mode of the RFID module

Parameters:

    Parameter: SearchMode

        Type - Byte

        Description - This parameter specifies the search mode (0) - dual or (1) - single to be set.

Return Value:

This function will return RFIDStatus.OK if it is able to set the RF module's search mode. It will return RFIDStatus.FAILED otherwise. This command will fail if a value other than 0 or 1 is passed in for the parameter.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

RFIDReader.SearchModeDual and RFIDReader.SearchModeSingle are provided as symbolic constants to make code using this interface more readable.

_____

RFIDStatus GetLinkParams(ref Byte LinkParams)

Function:

The method returns the current Link Parameter setting of the RFIDReader module.

Parameters:

    Parameter: LinkParams

        Type - ref Byte

        Description - The value of the Link Parameter setting will be returned through this parameter. The values returned will be:

        - 0 - DSB_ASK, FM0, 40 kHz

        - 1 - PR_ASK, Miller 4, 250 kHz

        - 2 - PR_ASK, Miller 4, 300 kHz

        - 3 - DSB_ASK, FM0, 400 kHz

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's link parameter. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

This method encapsulates the module command 0x54. See the "SensArray Communications Protocol, Part II" document for more details.

_____

RFIDStatus SetLinkParams(Byte LinkParams)

Function:

The method sets the current Link Parameter setting of the RFIDReader module.

Parameters:

Parameter: LinkParams

Type - Byte

Description - This function sets the link parameters to the following values:

- 0 - DSB_ASK, FM0, 40 kHz

- 1 - PR_ASK, Miller 4, 250 kHz

- 2 - PR_ASK, Miller 4, 300 kHz

- 3 - DSB_ASK, FM0, 400 kHz

Return Value:

This function will return RFIDStatus.OK if it is able to RF module's link parameters. It will return RFIDStatus.FAILED otherwise. This command will fail if the value passed in is not between 0 and 3.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

This method encapsulates the module command 0x52. See the "SensArray Communications Protocol, Part II" document for more details.

_____

RFIDStatus GetRegionSetting(out Byte RegionSetting)

RFIDStatus GetRegionSetting(out String RegionSettingString)

Function:

These methods return the region setting of the reader module. The initial overload returns a byte value for the region setting. The second overload returns a string representation.

Parameters:

Parameter: RegionSetting (first overload)
    Type – out Byte
    Description - Returns the current region setting of the reader module with values as follows:
    - 1 - China (840.5 – 844.5 MHz)
    - 2 - China 920.5 – 924.5 MHz)
    - 4 – ETSI/Europe
    - 8 – FCC/US/Canada/Mexico
    - 22 – Korea
    - 50 – Japan
    - 51 - Australia
    - 52 – Brazil
    - 53 – Europe
    - 54 – India
    - 55 – Israel
    - 56 - Taiwan
    - 50 – Japan
    - 22 – Korea
    - 70 - Bangladesh
    - 71 – Hong Kong
    - 72 - Indonesia
    - 73 – Malaysia
    - 74 – Singapore
    - 75 – Thailand
    - 76- Vietnam

Parameter: RegionSettingString (second overload)
    Type – out String
    Description - Returns a string representation of the current region setting of the reader module.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the RF module's region code or region name. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetModuleTemperature(ref Double ModuleTemperature)
Function:
This method retrieves the internal temperature of the RFIDReader module in degrees C.

Parameters:

    Parameter: ModuleTemperature

        Type - ref Double

        Description - On return, ModuleTemperature will hold the current internal temperature of the RFIDReader module in degrees C.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the internal temperature of the RFID module. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetConnectedAntennas(out Int32 AntennaCount, out Boolean[] AntennaIsConnected)

Function:

This method returns queries each of the antenna port of the integrated reader and returns an array sized appropriately for the reader of boolean (true/false) values indicating whether each of the port is connected or not.

Parameters:

    Parameter: AntennaCount

        Type - out Int32

        Description - Output specifying the number of antennas ports for the reader. For the current product offering, this number will be 1 for the 2-port reader and 4 for the 3-port reader.

    Parameter: AntennaIsConnected

        Type - out Boolean[]

        Description - The parameter returns an array of Boolean values. A "true" value indicates a reasonably matched antenna is attached. A "false" value indicates that an antenna is not attached. (However, see the notes below.)

Return Value:

This function will return RFIDStatus.OK if it is able to communicate with the reader and obtain information about whether an antenna is connected. RFIDStatus.FAILED will be returned otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

In addition to the external antenna ports, we return info regarding the internal port. Generally, the first element in the AntennaIsConnected array will return true.

We have found that if the antenna is in a place where there is a lot of metal in front of the antenna resulting in much of the transmitted RF power being reflected back at the antenna, this method will indicate that the antenna port is not connected even when an antenna is present.

Top-Level Sensor ID Queries

_____

RFIDStatus GetHardwareId(ref String HardwareId)

Function:

This method retrieves the hardware Major.Minor.Revision number of the sensor.

Parameters:

Parameter: HardwareId

Type - ref String

Description - Output returning the hardware identifier of the sensor.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the sensor's hardware version. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The value returned has the format V#.#.# where the first field (#) is the major version level of the hardware, the second the minor version level, and the third field contains the revision number.

_____

RFIDStatus GetFirmwareId(ref String FirmwareId)

Function:

This method retrieves the hardware Major.Minor.Revision number of the sensor.

Parameters:

Parameter: FirmwareId

Type - ref String

Description - Output returning the firmware version number of the sensor.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the sensor's firmware version. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The value returned has the format V#.#.# where the first field (#) is the major version level of the firmware, the second the minor version level, and the third field contains the firmware's revision number.

_____

RFIDStatus GetSerialNumber(ref String SerialNumber)

Function:
This method retrieves the hardware Major.Minor.Revision number of the sensor.
Parameters:
    Parameter: SerialNumber
        Type - ref String
        Description - Output returning the serial number of the sensor.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the sensor's serial number. It will return RFIDStatus.FAILED otherwise.
Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetFirmwareBuild(out String FirmwareBuild)

Function:
This method retrieves the reader firmware's build.
Parameters:
    Parameter: FirmwareBuild
        Type - out String
        Description - Output returning the firmware build number of the reader.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the sensor's build version. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established.

Additional Information:
The value returned has the format B.YY.MM.DD.RR where YY is the year of the build, MM is the month, DD is the day or the month, and RR is a sequence number if more than one version was developed on a given date. Note that initially, the first character of the build number is the letter "B". This may change for new readers if they have significantly different embedded firmware.

_____

RFIDStatus SetReaderDateAndTime(String ReaderDateAndTime)

Function:

Set the real-time clock date and time values of the reader

Parameters:

    Parameter: ReaderDateAndTime

        Type - String

        Description - This parameter provides the date/time to be set in the format: YYYY-MM-DDThh:mm:ss

Return Value:

This function will return RFIDStatus.OK if it is able to set the sensor's date and time. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetReaderDateAndTime(out String ReaderDateAndTime)

Function:

Retrieve the real-time clock date and time values from the reader

Parameters:

    Parameter: ReaderDateAndTime

        Type - out String

        Description - This parameter returns the reader's date and time in the format: YYYY-MM-DDThh:mm:ss

Return Value:

This function will return RFIDStatus.OK if it is able to get the sensor's current date and time. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetReaderConfig(out RFID.RFIDReader.ReaderConfig ReaderConfiguration)

Function:

Gets the configuration of the reader.

Parameters:

    Parameter: ReaderConfiguration

        Type - out RFID.RFIDReader.ReaderConfig ReaderConfig {

        Byte numberOfAntennas;
        Byte numberOfEthernetPorts;
        Byte numberOfGPIs;
        Byte numberOfGPOs; Boolean hasDCInput;
        }

        Description - Output parameter returns the hardware configuration of the reader.

Return Value:

This function will return RFIDStatus.OK if it is able to get the sensor's reader configuration. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

RFIDStatus SetReaderName(String ReaderName)

Function:

Sets the name advertised by the sensor's heartbeat messages.

Parameters:

    Parameter: ReaderName

        Type - String

        Description - Input parameter specifying the name of the reader to be advertised in the sensor's heartbeat message. The name cannot be longer than 31 characters.

Return Value:

This function will return RFIDStatus.OK if it is able to set the sensor's reader name. It will return RFIDStatus.FAILED otherwise. This command will fail if the reader name is longer than 31 characters.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetReaderName(out String ReaderName)

Function:

Gets the name advertised by the sensor's heartbeat messages through a direct firmware query.

Parameters:

    Parameter: ReaderName

        Type - out String

        Description - Output parameter returns the name of the reader that is advertised in the sensor's heartbeat message.

Return Value:

This function will return RFIDStatus.OK if it is able to get the sensor's reader name. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus GetReaderType(out String ReaderType)

Function:

Gets the reader type advertised by the sensor's heartbeat messages through a direct firmware query.

Parameters:

    Parameter: ReaderType

        Type - out String

Description - Output parameter returns the name of the reader that is advertised in the sensor's heartbeat message.

Return Value:
This function will return RFIDStatus.OK if it is able to get the sensor's reader type. It will return RFIDStatus.FAILED otherwise.
Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Networking and Other Sensor-Level Configuration Functions

_____

RFIDStatus GetIpV4Info(ref Boolean UsingDHCP,
ref Byte[] IPv4Address, ref Byte[] ipv4Netmask,
ref Byte[] ipv4Gateway, ref Byte[] ipv4DNSServer,
ref UInt16 SensorPort)

Function:
This method returns the various parameters comprising the IpV4 network configuration.
Parameters:
Parameter: UsingDHCP
Type - ref Boolean
Description - Output whose value is True if the sensor is getting its IPv4 setup using DHCP, False if a static IP address has been assigned.

Parameter: IPv4Address, ipv4Netmask, ipv4Gateway, ipv4DNSServer
Type - ref Byte[4]
Description - These four output values contain the IP addresses for the associated networking parameters. In dotted notation, if the address is of the form a.b.c.d, then val[0] = a, val[1] = b, val[2] = c, and val[3] = d.

Parameter: SensorPort
Type - ref UInt16
Description - On return, this output contains the value of the TCP port used for command-level communication with the sensor. takes place over.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the sensor's IpV4 network configuration. It will return RFIDStatus.FAILED otherwise.
Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus SetIpV4Info(Boolean UseDHCP,

  Byte[] IPv4Address, Byte[] ipv4Netmask,

  Byte[] ipv4Gateway, Byte[] ipv4DNSServer,

  UInt16 SensorPort)

Function:

This method is used to set the IpV4 network configuration for the sensor.

Parameters:

   Parameter: UseDHCP

      Type - Boolean

      Description - Input specifying that the sensor should use DHCP for to obtain its network configuration.

   Parameter: IPv4Address, ipv4Netmask, ipv4Gateway, ipv4DNSServer

      Type - String

      Description - Input values for IpV4 address, the netmask, the gateway's IP address, and the IP address of the DNS server. These should be a valid dot notation for these values (a.b.c.d where the 4 fields must fall between 0 and 255).

   Parameter: SensorPort

      Type - UInt16

      Description - Input specifying the port for command-level communication with the sensor. This value must be between 1024 and 65535.

Return Value:

This function will return RFIDStatus.OK if it is able to set the sensor's IpV4 networking setup. It will return RFIDStatus.FAILED otherwise. This method will fail if the SensorPort number is not between 1024 and 65535.

Exception(s) thrown:

- A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

- A System.FormatException will be thrown if any of the parameters IPv4Address, ipv4Netmask, ipv4Gateway, or Ipv4DNSServer is not a valid dot notation IP address

Additional Information:

Note that this command will immediately change the network setup, then close and reopen the command socket. Also, this configuration will not be automatically saved as the startup configuration of the sensor. Consequently, if the sensor is rebooted before the sensor's configuration is saved it will restart using the previous sensor setup.

_____

RFIDStatus GetHeartbeatConfig(ref Byte[] IPv4Address, ref UInt16 PortNumber,
ref UInt32 Interval, ref UInt32 Count)

Function:

This method returns the IP address and UDP port number over which the heartbeat is sent as well and the Interval between heartbeat messages and the number (Count) of messages that are sent before the heartbeat goes silent.

Parameters:

Parameter: IPv4Address

Type - ref Byte[]

Description - This output parameter returns with the four bytes that comprise the IP address over which the heartbeat messages are sent.

Parameter: PortNumber

Type - ref UInt16

Description - Output parameter returning the UDP port number over which the heartbeat messages are being sent.

Parameter: Interval

Type - ref UInt32

Description - On return this output parameter will hold the time interval in seconds between heartbeat messages.

Parameter: Count

Type - ref UInt32

Description - On return this output parameter contains the number of counts until the heartbeat is silenced.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the heartbeat configuration. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus SetHeartbeatConfig(Byte[] IPv4Address, UInt16 PortNumber,
UInt32 Interval, UInt32 Count)

Function:

This method sets the IP address and UDP port number over which the heartbeat is sent as well and the Interval between heartbeat messages and the number (Count) of messages that are sent before the heartbeat goes silent.

Parameters:

    Parameter: IPv4Address

        Type - Byte[]

        Description - This input parameter passes in the four bytes that comprise the IP address over which the heartbeat messages are sent. This can be set to 255.255.255.255 for UDP broadcast over the local subnet.

    Parameter: PortNumber

        Type - UInt16

        Description - This input parameter sets the PortNumber over which the UDP heartbeat messages are sent. The default is 3988 but can be set to any value between 1024 and 65535.

    Parameter: Interval

        Type - UInt32

        Description - The Interval input parameter passes in the number of seconds between heartbeat messages. This value can be set to 0 to turn the heartbeat off entirely, or can be set to a long interval to reduce network traffic associated with the heartbeat messages.

    Parameter: Count

        Type - UInt32

        Description - This value specifies the number of heartbeat messages that are sent before the heartbeat is no longer being sent. Again, this can be used to reduce network traffic by turning the messages off after some time period.

Return Value:

This function will return RFIDStatus.OK if it is able to set the heartbeat configuration. It will return RFIDStatus.FAILED otherwise. This command will fail if the PortNumber is not between 1024 and 65535.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The Interval and Count values should be used carefully since it could make device network discovery difficult. During initial deployment, you may want to leave these at their default values (Interval=30, Count=0xFFFFFFFF) or even consider reducing the interval to a smaller value so that you see the heartbeat more often.

Also, note that this configuration is not saved automatically. Be sure to save the configuration so that the new values will stay in effect when the sensor is rebooted.

_____

RFIDStatus GetLocatorSignalStatus(ref Boolean LocatorSignalActive)

Function:

This method returns whether or not the locator flash pattern is active on the sensor.

Parameters:

 Parameter: LocatorSignalActive
  Type - ref Boolean
  Description - Output is true when the locator signal is active, false otherwise.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the reader's locator signal status. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The LED on the sensor blinks alternating red and green when the signal locator is on to help locate a specific sensor.

_____

RFIDStatus SetLocatorSignalStatus(Boolean NewSignalStatus)

Function:

This method activates or deactivates the locator signal on the sensor.

Parameters:

 Parameter: NewSignalStatus
  Type - Boolean
  Description - Input parameter, when true, turns the locator flash pattern on, and when false, returns the flash pattern to normal operating mode indication.

Return Value:

This function will return RFIDStatus.OK if it is able turn the reader's locator indicator on or off. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The LED on the sensor blinks alternating red and green when the signal locator is on. It returns to its standard blinking green light when the locator is turned back off.

_____

RFIDStatus GetTempNotificationSetup(ref UInt16 notificationInterval, ref UInt16 alertInterval,
ref Double warningThreshold, ref Double alertThreshold)

Function:
This method returns the time intervals and thresholds for reporting module temperature during continuous tag read sessions.

Parameters:
Parameter: NotificationInterval
Type - ref UInt16
Description - Output value returning the time interval between UDP notifications of the temperature of the module.

Parameter: AlertInterval
Type - ref UInt16
Description - Output returning the time interval between times that the sensor checks the module's temperature to determine whether it is over the warning or the alert threshold.

Parameter: WarningThreshold
Type - ref Double
Description - Output parameter that returns the current temperature threshold in degrees C above which a warning notification is sent.

Parameter: AlertThreshold
Type - ref Double
Description - Output parameter that returns the current temperature threshold in degrees C above which a temperature alert notification is sent.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the RF module's temperature notification settings. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
See the comments under the SetTempNotificationSetup() method documentation for more details.

_____

RFIDStatus SetTempNotificationSetup(UInt16 notificationInterval, UInt16 alertInterval, Double warningThreshold, Double alertThreshold)

Function:

This method sets the time intervals and thresholds for reporting module temperature during continuous tag read sessions.

Parameters:

    Parameter: NotificationInterval

        Type - UInt16

        Description - Input value specifying the time interval in seconds between UDP notifications of the temperature of the module. Setting this value to 0 turns this notification off.

    Parameter: AlertInterval

        Type - UInt16

        Description - Input specifying the interval in seconds between times that the sensor checks the module's temperature to determine whether or not it is over the warning or the alert threshold.

    Parameter: WarningThreshold

        Type - Double

        Description - Input parameter to set the current temperature threshold in degrees C above which a warning notification is sent.

    Parameter: AlertThreshold

        Type - Double

        Description - Input parameter to set the current temperature threshold in degrees C above which a temperature alert notification is sent.

Return Value:

This function will return RFIDStatus.OK if it is able to set the reader's temperature notification parameters. It will return RFIDStatus.FAILED otherwise. This method will fail if an attempt is made to set the alert interval to less than 10 seconds or the alarm threshold is set above 85 degrees C.

Exception(s) thrown:

A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

There are two types of UDP notifications to port 3984 that are sent from the sensor related to temperature.

The first is a periodic notification of the module's temperature at an interval specified by the NotificationInterval parameter. This can be turned off by setting it to 0. The default for this interval is 30 seconds.

These notifications are sent when the module exceeds a couple of thresholds. A warning is issued when the temperature exceeds the value specified by the WarningThreshold parameter. An alert is issued when the temperature exceeds the value specified by AlertThreshold. As long as the temperature stays above these values, the warning or the alert is not resent. However, if the temperature drops at least 2 degrees below these levels, then rises above them again, the warning or alert is reissued.

The warnings can be turned off by setting the WarningThreshold very high. However, to protect the module, alerts cannot be turned off and the threshold cannot be set above 85 degrees C.

The default warning threshold is 60 degrees C. The default alert threshold is 85 degrees C. These thresholds are checked periodically determined by the AlertInterval parameter. The time interval has a default value of 30 seconds. It cannot be se lower than 10 seconds. If this value is set to a very long time interval (> 5 minutes) the module may become excessively hot and will automatically shut down to protect itself.

Bluetooth and Wi-Fi Configuration Functions

RFIDStatus GetWiFiConfiguration(out String SSID)

Function:

This method retrieves the name of the wireless network (SSID) for the Wi-Fi router that the SensX Extreme reader uses for Wi-Fi connections.

Parameters:

    Parameter: SSID

        Type - out String

        Description - This parameter returns the name of the wireless network the reader uses to connect to your wireless router. Note that this method will return an empty string if the SSID has not been set.

Return Value:

This method will return RFIDStatus.OK if the SSID can be retrieved, otherwise, the method will return RFIDStatus.Failed.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the reader doesn't respond.

_____

RFIDStatus GetBluetoothConfiguration(out String Address)

Function:

This method retrieves the address of the Bluetooth device that the reader is configured to send data to.

Parameters:

Parameter: Address

Type - out String

Description - This parameter returns the Bluetooth device the reader will send data to. The address returned will be a string formatted as ##:##:##:##:##:## where '#' represents a hexidecimal digit (0-9,A-F). Note that this method will return an empty string (length 0) if the Bluetooth address has not been set on your reader.

Return Value:

This method will return RFIDStatus.OK if the Address can be retrieved, otherwise, the method will return RFIDStatus.Failed.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the reader doesn't respond.

_____

RFIDStatus SetWiFiConfiguration(String SSID, String Passcode)

Function:

This method configures the Wi-Fi parameters in the SensX Extreme readers to enable connections to a Wi-Fi router. The name of the wireless network (SSID) and the password (Passcode) needed to access that network are required for the reader to communicate on that wireless network. The SensX reader requires the wireless router to support WPA2_Personal_AES security settings.

Parameters:

Parameter: SSID

Type - String

Description - This parameter is used to specify the name of the wireless network as is typically broadcast by your wireless router.

Parameters:

Parameter: Passcode

Type - String

Description - This parameter is required to allow the reader to provide the password or passphrase to the wireless router to allow it onto the wireless network.

Return Value:

This method will return RFIDStatus.OK if the SSID and Passcode are not empty strings. If one or the other of these parameters is an empty String, the method will return RFIDStatus.Failed.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the reader doesn't respond.

_____

RFIDStatus SetBluetoothConfiguration(String Address, String PairingCode)

Function:

This method configures the reader to establish a connection with a specific Bluetooth enabled device such as a Smart Phone, a tablet computer, etc. This communication is enabled by providing the reader with the address and the pairing code of the device that the reader will pair with and subsequently communicate with.

Parameters:

    Parameter: Address

        Type - String

        Description - This parameter specifies the Bluetooth Address of the device that will be paired with the reader. The Address must be a string in the form ##:##:##:##:##:## where each pair of ## fields specifies a hexadecimal value. An example of this might be 11:22:34:56:78:90. Note that there are 6 fields separated by the colon ':' character. This method requires that each # character be one of the values 0-9 or A-F. Also, leading zeros must be provided -- e.g., 01:02:03:04:05:06 cannot be shortened to 1:2:3:4:5:6.

    Parameter: PairingCode

        Type - String

        Description - The PairingCode is the security code required by the device that the reader will pair with. This code must be passed by the reader to the Bluetooth-enabled device to provide the credentials that allow the Bluetooth device to know that the reader is authorized to connect. Note that this can be a UTF-8 encoded string.

Return Value:

This method will return RFIDStatus.OK if the Address and PairingCode are properly formatted as specified in the parameter descriptions above. If either of these parameters is not properly formatted, this method will return RFIDStatus.Failed.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the reader doesn't respond.

Configuration Save and Restore

_____

RFIDStatus SaveCurrentConfiguration()

Function:
This method saves the various configurable sensor settings to flash memory so that they will be active when the sensor reboots.

Return Value:
This function will return RFIDStatus.OK if it is able to save the settings. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
The configuration information saved includes:
- The Reader Name
- Networking parameters
- Heartbeat parameters
- Antenna Power values
- Antenna Sequencing setup
- Temperature Management parameters

_____

RFIDStatus RestoreSavedConfiguration()

Function:
This method sends a request to the sensor to restore the saved configuration values that are stored in flash memory.

Return Value:
This function will return RFIDStatus.OK if it is able to read and restore the reader's saved configuration. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
This command restores the configuration information documented under the comments section of the SaveCurrentConfiguration() method. Note that the sensor does not check whether the networking parameters are reset. Consequently, the sensor shuts down the TCP

communication socket and reopens it to wait for new commands. The underlying communications layer for this API handles this situation and also works to reestablish communication. Your application layer, however, will need to provide the IP address and port number to reestablish this communication link if these values are different between the saved configuration and the setup prior to calling this method.

The primary purpose of this command is to allow you to experiment with different configurations, then elect to restore the last saved configuration if things don't behave the way you want. (This can also be achieved by rebooting or power-cycling the sensor.)

_____

RFIDStatus ResetToDefaultConfiguration()

Function:
This resets the various sensor configuration parameters back to their default values, then restarts the TCP/IP stack.

Return Value:
This function will return RFIDStatus.OK if it is able to reset the reader's configuration . It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
This command restores the configuration information documented under the comments section of the SaveCurrentConfiguration() method back to its factory defaults. Once the defaults have been restored and the sensor acknowledges having received the command, it shuts down the TCP communication socket and reopens it to wait for new commands. Your application layer will need to provide the IP address and port number to reestablish to this class so that it can attempt to open communication on the right IP address and port.

_____

RFIDStatus Reboot()

Function:
This method issues the reboot command to the sensor.

Return Value:
This function always returns RFIDStatus.OK.

Exception(s) thrown:
A System.TimeoutException will be thrown if a connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

DCN-ST-00001-D 15 November 2021

Note that as long as the sensor receives the command issued by this method, it will reboot immediately without sending a response. This method will close the underlying socket. Consequently, it is the responsibility of the application to provide the correct IP address and port number to this class if the saved configuration is different from the current setup.

_____

RFIDStatus GetBootloaderInfo(ref UInt16 bootloaderPort)

RFIDStatus GetBootloaderInfo(out UInt16 bootloaderPort,
out int versionMajor, out int versionMinor)

Function:
These methods return the Ethernet port over which firmware updates are performed. The second interface also returns the major and minor version numbers of the underlying bootloader code. If your application simply needs to know what port is configured for communicating with the bootloader, use the first call.

Parameters:
　　Parameter: bootloaderPort
　　　　Type - out Uint16
　　　　Description - This parameter returns the UDP Ethernet port over which the application firmware is communicated to the reader.

　　Parameter: versionMajor, versionMinor
　　　　Type - out int, out int
　　　　Description - These return the major and minor version numbers of the installed bootloader.

Return Value:
This function will return CommandStatus.OK if it is able to read and return the bootloader Ethernet communication port and the version information. It will return CommandStatus.FAILED otherwise.

Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

GPIO and 24V Management Methods

_____

RFIDStatus GetGPIOSetup(ref Byte GPIOSetup)

Function:

This method returns the settings for the general-purpose outputs and the detected state of the general-purpose inputs

Parameters:

    Parameter: GPIOSetup

        Type - ref Byte

        Description - This output parameter returns the settings of the general-purpose inputs and outputs. The 8 bits of the value are [i4, i3, i2, i1, o4, o3, o2, o1]. A 1 for the input values indicating a high voltage level (between 5 and 24 volts) and a 0 indicating a level below 3 volts. A 1 for an output bit indicates that the output has been turned on, and a 0 indicates it is off.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the GPIO setting. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus SetGeneralPurposeOutputs(Byte NewGPOs)

Function:

This method sets the settings for the general-purpose outputs and the detected state of the general-purpose inputs

Parameters:

    Parameter: NewGPOs

        Type - Byte

        Description - This input parameter is sent to the sensor to turn the general-purpose outputs on or off. The 8 bits of the value are [0, 0, 0, 0, o4, o3, o2, o1]. A 1 turns the output on, and a 0 turns it off.

Return Value:

This function will return RFIDStatus.OK if it is able to set the new output values. It will return RFIDStatus.FAILED otherwise. This command will fail if the 4 most significant bits are not set to 0.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus Get24VStatus(ref Boolean V24State, ref Boolean V24OnAtStartup)

Function:

This method returns the current setup of the 24V DC output on the GPIO connector as well as what its state is when the sensor is started or restarted. Not applicable for Extreme model.

Parameters:

    Parameter: V24State

        Type - ref Boolean

        Description - This output parameter is true if the 24VDC power is turned on, and false when it is turned off.

    Parameter: V24OnAtStartup

        Type - ref Boolean

        Description - This output parameter is true if the 24VDC is automatically turned on when the sensor is started or restarted and false if it is off at startup.

Return Value:

This function will return RFIDStatus.OK if it is able to set the status of the 24V DC power supply. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus Set24VStatus(Boolean V24Active, Boolean V24ActiveOnStartup)

Function:

This method turns the 24VDC on or off and sets the configuration that determines whether it is on or off on startup. Not applicable for Extreme model.

Parameters:

    Parameter: V24Active

        Type - Boolean

        Description - This input parameter turns the 24VDC power on when true and off when false.

    Parameter: V24ActiveOnStartup

        Type - Boolean

        Description - This input parameter determines whether the 24VDC power is on (true) when the sensor is restarted, or off (false).

Return Value:

This function will return RFIDStatus.OK if it is able to change to the new setup for the 24VDC. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The initial configuration of the 24VDC is set via V24ActiveOnStartup. However, as with any configuration item that is part of the startup process, the method SaveCurrentConfiguration() method needs to be called for it to be in effect when the reader is booted.

Continuous Inventory Setup and Control

_____

RFIDStatus GetReadGapTimes(ref UInt16 ReadTime, ref UInt16 GapTime)

Function:

This command retrieves the current read time interval and the time interval between read cycles.

Parameters:

    Parameter: ReadTime

        Type - ref UInt16

        Description - Output parameter containing the time interval in seconds for which a inventory read cycle is performed for each antenna driven by the reader.

    Parameter: GapTime

        Type - ref UInt16

        Description - Output parameter containing the time interval in milliseconds for the time period between inventory read cycles.

Return Value:

This function always RFIDStatus.OK.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus SetReadGapTimes(UInt16 ReadTime, UInt16 GapTime)

Function:

This method sets the time interval for an inventory read cycle and the time interval between reads.

Parameters:

    Parameter: ReadTime

        Type - UInt16

Description - Input parameter setting the time interval in milliseconds for which the reader will perform an inventory cycle for the current antenna.

Parameter: GapTime
Type - UInt16
Description - Input parameter setting the time interval in milliseconds between inventory read cycles.

Return Value:
This function will return RFIDStatus.OK if it is able to set the read time and the gap time of the sensor. It will return RFIDStatus.FAILED otherwise. This command will fail if an attempt is made to set the GapTime to less than 10 ms.

Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:
The read time is the time interval for reading tags for each antenna in the antenna sequence. The gap time is the time interval between read cycles where the reader switches to the next antenna in the sequence, sets up the read power level for driving that particular antenna as well as doing a bit of housekeeping. The minimum time interval allotted for these to complete is 10 ms, hence the minimum value validation for the gap time.

Note that the read and gap times establish a duty cycle for the operation of the RF module. Since the module can overheat when run continuously, we encourage you to either monitor the module's temperature carefully during high duty cycle operation, or set the duty cycle up to allow for extended operation under your ambient temperature conditions.

More information about selecting appropriate read and gap times based on ambient temperature can be found in the SensThys white paper "Thermal Performance of the SensArray".

_____

RFIDStatus GetReadSequence(ref Byte[] AntennaSequence, ref Byte SequenceLength)

Function:
This method retrieves the sequence of antennas that the reader loops through when performing a continuous tag inventory read cycle.

Parameters:
Parameter: AntennaSequence
Type - ref Byte[]
Description - This output parameter returns an array of antenna numbers reflecting the sequence of antennas that the reader will cycle through during a continuous inventory read cycle.

Parameter: SequenceLength
Type - Byte

Description - Output value indicating the number of items in the antenna sequence.

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the RF module's hardware id. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

Note that the antenna sequence can have repeats. For example, the sequence [0, 0, 1, 0, 1, 1] (with SequenceLength 6) would read antenna 0 for 2 cycles, then alternate between antenna 1 and antenna 0 for one cycle each, then read antenna 1 for 2 consecutive cycles before looping back and repeating the entire sequence.

The purpose of this is to allow for effectively longer read periods on some antennas than on others due to tag density or other factors that might effect read accuracy.

_____

RFIDStatus SetReadSequence(Byte[] AntennaSequence, Byte SequenceLength)

Function:

This method sets the sequence of antennas that the reader loops through when performing a continuous tag inventory read cycle.

Parameters:

    Parameter: AntennaSequence

        Type - Byte[]

        Description - This input parameter is used to pass an array of antenna numbers to the reader to set the sequence of antennas that it will cycle through during a continuous inventory read cycle.

    Parameter: SequenceLength

        Type - Byte

        Description - Input value indicating the number of items in the antenna sequence.

Return Value:

This function will return RFIDStatus.OK if it is able to set the reader's antenna sequence. It will return RFIDStatus.FAILED otherwise. This command will fail if the antenna numbers in the sequence fall outside the range from 0 to 3. Also, the maximum number of antennas is 16, this method will fail if SequenceLength is greater than 16.

Exception(s) thrown:

- A System.IndexOutOfRangeException will be thrown if the SequenceLength is larger than the number of elements the AntennaSequence array.

- A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

Note that the antenna sequence can have repeats. For example, the sequence [0, 0, 1] (with SequenceLength 3) would read antenna 0 for 2 cycles, read antenna 1, then loop back repeating the entire sequence. This would effectively give antenna 0 twice as much air time as antenna 1.

The purpose of this is to allow for effectively longer read periods on some antennas than on others due to tag density or other factors that might affect read accuracy.

_____

RFIDStatus SetInventoryFilter(Byte maskBank,
UInt16 maskStartBitAddress,
UInt16 maskBitLength,
Byte[] mask,
Boolean makePersistent)

Function:

This method sets the filter to be used during tag inventory cycles.

Parameters:

    Parameter: maskBank, maskStartBitAddress, maskBitLength, mask

        Type - UInt8, UInt16, UInt16, Byte[]

        Description - These 4 fields specify a mask for filtering which tags to read. The maskBank specifies the data bank to filter on (1 = EPC data, 2 = Tag Identifier data, or 3 = User data). The starting bit position within the data bank that the filter begins from is specified by maskStartBitAddress, the length of the mask in bits is specified by maskBitLength, and the actual mask data is specified by the mask parameter.

        As an example, if you wanted to mask based on the TID of a tag with TID value E20034140117010112DD127E, maskBank = 2, maskStartBit = 0, maskBitLength = 96 (12 bytes * 8 bits/byte) and mask = [E2, 00, 34, 14, 01, 17, 01, 01, 12, DD, 12, 7E].

    Parameter: makePersistent

        Type - Boolean

        Description - This parameter specifies whether this filter should be saved to the module's flash memory so that it will continue to be active even when the module reboots.

Return Value:

This function will return RFIDStatus.OK if it is able to set the filter. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

Inventory filters can be cleared by calling SetInventoryFilter(0, 0, 0, null, makePersistent). We have also provided the method ClearInventoryFilter(makePersistent) as a convenience wrapper for this.

_____

RFIDStatus ClearInventoryFilter(Boolean makePersistent)

Function:

This method clears the filter used during tag inventory cycles.

Parameters:

    Parameter: makePersistent

        Type - Boolean

        Description - This parameter specifies whether any filter saved in the module should be permanently cleared. If set to false, the next time the module reboots, the prior saved filter will become active again.

Return Value:

This function will return RFIDStatus.OK if it is able to clear the filter. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

This is a convenience method that simply calls SetInventoryFilter(0, 0, 0, null, makePersistent).

_____

RFIDStatus StartContinuousRead()

Function:

This method starts a continuous tag inventory session. This session continues until the StopContinuousRead() command is received. The "Additional Information" section below describes in more detail how to use these calls.

Return Value:

This function always returns RFIDStatus.OK.

Exception(s) thrown:

A System.TimeoutException will be thrown if the initial connection to the sensor cannot be established.

Additional Information:

This method along with ReadNextTag() – or your registered callback – and StopContinuousRead() form a set of commands for performing a continuous tag inventory read cycle. During a continuous read cycle, the reader automatically cycles through the antenna sequence as described in the comments sections of the SetReadGapTimes() and SetAntennaSequence() commands above.

There are two ways to use these commands to perform the inventory cycle:

- The first way to use these commands is to register a callback method with the RFIDReader class, either through the class constructor, or through the SetTagReadCallback() method. The callback interface is documented in the constructor section above. Whenever a callback is registered, StartContinuousRead() starts a background thread to read the tag data continuing until the StopContinuousRead() command is sent. Due to network and process latency, when StopContinuousRead() is sent from your application, the background thread continues to read tags until the response to the StopContinuousRead() command is received. At that point, the status data member of the TagReadInfo structure passed back as the callback parameter is set to RFIDStatus.DONE and the background thread stops reading tags.
- The second way to use these commands is to issue StartContinuousRead(), then invoke ReadNextTag() in a loop until the return value from the method call is RFIDStatus.DONE. RFIDStatus.DONE is issued from ReadNextTag() when it receives the response to the StopContinuousRead() command rather than tag data. Note that due to network latency and other factors, a number of tags may be read after the stop command is issued. Because ReadNextTag() blocks indefinitely waiting for tag data to be returned, StopContinuousRead() must be issued from a separate thread. Typically, a thread would be created to run the ReadNextTag() loop, and the StopContinousRead() command can then be issued from the primary thread.

In either scenario, once the StartContinuousRead() method is invoked, the only command that should be sent to the sensor is StopContinuousRead(). ReadNextTag() can be called to listen for the data from the next tag read to be passed back. Otherwise, any other command can result in the sensor locking up and leaving the module in a continuous read state, potentially damaging the reader.

_____

RFIDStatus ReadNextTag(ref String EPC, ref Double RSSI, ref Byte AntennaNumber)

Function:
Summary
Parameters:
    Parameter: EPC
        Type - ref String
        Description - This output parameter returns the EPC for the first tag read after the continuous read cycle is started.

    Parameter: RSSI
        Type - ref Double
        Description - RSSI is an output parameter returning the detected RSSI in dBm for the tag that was read.

Return Value:

This method returns RFIDStatus.DONE when the response from StopContinuousRead() has been received. This function will generally return RFIDStatus.OK. RFIDStatus.FAILED will be returned if malformed data is read.

Exception(s) thrown:

Generally, no exceptions will be thrown for this call. As always, it is good practice to put this in a try/catch block to handle unusual situations that might crop up and cause your application to crash if not caught.

Additional Information:

See the discussion under the Additional Information section for the StartContinuousRead() method for how to perform a continuous tag inventory cycle.

Note that this command blocks waiting for the next tag to be returned. Because of this, the only way to stop the continuous read cycle is to call the StopContinousRead() from a separate thread.

_____

RFIDStatus StopContinuousRead()

Function:

This method is called to stop a continuous read cycle started by the StartContinuousRead() method.

Return Value:

This always returns RFIDStatus.OK.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor has been lost and cannot be reestablished in a timely manner.

Additional Information:

See the discussions under the Additional Information sections of the StartContinuousRead() method and ReadNextTag() method above.

_____

RFIDStatus ResetReceiveTimeout()

Function:

This is a cleanup command that needs to be called after a continuous read cycle has been stopped. StartContinuousRead() sets the receive timeout to be infinite so that the ReadNextTag() call will block waiting for the next tag data to be sent.

Return Value:

This function always returns RFIDStatus.OK.

Exception(s) thrown:

No exceptions will be thrown.

_____

RFIDStatus ReadTag(ref String EPC, ref Double RSSI, ref Byte AntennaNumber)

Function:

This method requests a single inventory read cycle returning any detected tag.

Parameters:

    Parameter: EPC
        Type - ref String
        Description - Output parameter returning the EPC code for the tag read.

    Parameter: RSSI
        Type - ref Double
        Description - This output parameter returns the RSSI value of the read tag in dBm.

    Parameter: AntennaNumber
        Type - ref Byte
        Description - Output parameter returning the antenna number that was energized during the tag inventory cycle.

Return Value:

This function will return RFIDStatus.OK if it is able to read a tag. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

This command will return "No Data" in the EPC field if it returns without reading a tag. It will return "Bad Data" if the data read is corrupted.

Tag Commissioning and Decommissioning

_____

RFIDStatus ReadTagData(UInt32 accessPassword,
        Byte maskBank, UInt16 maskStartBit, UInt16 maskBitLength, Byte[] mask,
        Byte dataBank, UInt16 dataStartWord, UInt16 dataWordLength, ref UInt16[]
        data,
        out Byte errorCode)

Function:

This method reads the tag data from the specified tag data bank: 0 = Reserved data, 1 = EPC data, 2 = Tag Identifier Data (TID), 3 = User data. The tags read are filtered using the mask data against the location specified by the maskBank, maskStartBit (offset into bank in number of bits) with length specified by maskBitLength number of bits. DataWordLength number of words will be read from the bank specified by the dataBank parameter starting at offset dataStartWord words.

Parameters:

    Parameter: accessPassword

Type - UInt32

Description - This specifies the password needed when reading the tag data. In general, specifying a non-zero value is only needed when the access password and kill password have been locked. For all other data banks and to read the passwords when they are not locked, either 0 or the value that matches the currently programmed password for the tag can be used.

Parameter: maskBank, maskStartBit, maskBitLength, mask

Type - UInt8, UInt16, UInt16, Byte[]

Description - These 4 fields specify a mask for filtering which tag is to be read. The maskBank specifies the data bank to filter on (1 = EPC data, 2 = Tag Identifier data, or 3 = User data). The starting bit position within the data bank that the filter begins from is specified by maskStartBit, the length of the mask in bits is specified by maskBitLength, and the actual mask data is specified by the mask parameter.

As an example, if you wanted to mask based on the TID of a tag with TID value E20034140117010112DD127E, maskBank = 2, maskStartBit = 0, maskBitLength = 96 (12 bytes * 8 bits/byte) and mask = [E2, 00, 34, 14, 01, 17, 01, 01, 12, DD, 12, 7E].

Parameter: dataBank, dataStartWord, dataWordLength, data

Type - UInt8, UInt16, UInt16, ref UInt16[]

Description - The parameter dataBank specifies the data bank to be read (0 = Reserved data, 1 = EPC data, 2 = Tag Identifier data, 3 = User data), dataStartWord specifies what word (2 bytes) boundary to start from, and dataWordLength specifies the number of words to read. The data is returned as dataWordLength number of elements filled in the data array. Note that the data array is allocated by the calling program and must be at least dataWordLength elements long.

As an example, if we wanted to read words 2 and 3 of the TID specified above, we would pass in dataBank=2, dataStartWord = 1 (the indexing is zero based) and dataWordLength = 2. On a successful read, this method would return data = [3414, 0117].

Parameter: errorCode

Type - out Byte

Description - This returns the underlying error code detailing any problems. Typical error codes are: 1 - the specified parameters are not supported, 2 - the password specified was wrong or provided insufficient privileges (e.g., 0 when associated parts of the tag are locked, and 3 - mask or data locations and/or lengths were not specified correctly. These error codes correspond to the values documented in Annex I of the EPCGlobal Gen2 Specification (Nov-2013, Version 2.0).

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the data requested for the given tag. It will return RFIDStatus.FAILED otherwise and set the errorCode parameter indicating the specific error condition.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond. A System.IndexOutOfRangeException will be thrown if the "data" array was not allocated with at least a size of dataWordLength.

_____

RFIDStatus WriteTagData(UInt32 accessPassword,

> *Byte maskBank, UInt16 maskStartBit, UInt16 maskBitLength, Byte[] mask,*
> *Byte dataBank, UInt16 dataStartWord, UInt16 dataWordLength, UInt16[] data,*
> *out Byte errorCode)*

Function:

This method writes the tag data to the specified tag data bank: 0 = Reserved data, 1 = EPC data, 2 = Tag Identifier data (generally not writable), 3 = User data using the mask data against the location specified by the maskBank, maskStartBit (offset into bank in number of bits) with length specified by maskBitLength number of bits. DataWordLength number of words will be written to the bank specified by the dataBank parameter starting at offset dataStartWord words.

Parameters:

The parameters for this method are described in detail under the ReadTagData() method above. The only exception is that the data[] array is passed in and is written to the specified location (rather than read from that location in the description above).

Return Value:

This function will return RFIDStatus.OK if it is able to read and return the data requested for the given tag. It will return RFIDStatus.FAILED otherwise. and set the errorCode parameter indicating the specific error condition. These error codes are documented under the ReadTagData/errorCode parameter section above.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus LockTagData(UInt32 accessPassword,

> *Byte maskBank, UInt16 maskAddress, UInt16 maskLength, Byte[] mask,*
> *RFIDLockState killPwdLock, RFIDLockState accessPwdLock,*
> *RFIDLockState epcLock, RFIDLockState userDataLock,*
> *out Byte errorCode)*

Function:

This method locks, unlocks, or permanently locks or unlocks one or more data banks on the tag. The tag to which the lock is to be applied can be selected using the mask data.

Parameters:

Parameter: accessPassword
    Type - UInt32
    Description - This specifies the password needed to lock the tag data.

Parameter: maskBank, maskStartBit, maskBitLength, mask
    Type - UInt8, UInt16, UInt16, Byte[]
    Description - These 4 fields specify a mask for filtering which tag is to be read. The maskBank specifies the data bank to filter on (1 = EPC data, 2 = Tag Identifier data, or 3 = User data). The starting bit position within the data bank that the filter begins from is specified by maskStartBit, the length of the mask in bits is specified by maskBitLength, and the actual mask data is specified by the mask parameter.
    As an example, if you wanted to lock the tag whose TID is E20034140117010112DD127E, you would pass in maskBank = 2, maskStartBit = 0, maskBitLength = 96 (12 bytes * 8 bits/byte) and mask = [E2, 00, 34, 14, 01, 17, 01, 01, 12, DD, 12, 7E].

Parameter: killPwdLock, accessPwdLock, epcLock, userDataLock
    Type - RFIDLockState
    Description - These parameters specify the type of locks that are to be applied to each of the respective data items. Each of these lock parameters is the or (|) of the following values:
    • RFIDLocKState.NoChange = 0 -- The lock state of the associated bank is not be changed.
    • RFIDLocKState.Lock = 1 – The bank's lock bit is to be set.
    • RFIDLockState.Unlock = 2 – The bank's lock bit is to be cleared
    • RFIDLocKState.PermaLock = 4 – The bank's permalock bit is to be set.
    • RFIDLockState.PermaUnlock = 8 – The bank's permalock bit is to be cleared.

    RFIDStatus.FAILED is returned if Lock and Unlock are both specified or PermaLock and PermaUnlock are both specified. Also, errorCode will be set to 0 (non-specified error code under the EPCglobal Gen2 Specification.
    The response of the tag to read and write commands when these various bits are set of cleared can be found on page 89 of the EPCglobal Gen2 Specification (Nov-2013, Version 2.0).

Parameter: errorCode
    Type - out Byte
    Description - This returns the underlying error code detailing any problems. Typical error codes are: 1 - the specified parameters are not supported, 2 - the password

specified was wrong or provided insufficient privileges (e.g., 0 when associated parts of the tag are locked, and 3 - mask or data locations and/or lengths were not specified correctly. These error codes correspond to the values documented in Annex I of the EPCglobal Gen2 Specification (Nov-2013, Version 2.0).

Return Value:

This function will return RFIDStatus.OK if it is able to change the lock status of all of the specified data banks. It will return RFIDStatus.FAILED otherwise. and set the errorCode parameter indicating the specific error condition. In addition, none of the specified locks are applied. The values for the returned error code are documented under the ReadTagData/errorCode parameter section above.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

_____

RFIDStatus KillTag(UInt32 killPassword,
Byte maskBank, UInt16 maskAddress, UInt16 maskLength, Byte[] mask,
out Byte errorCode)

Function:

This method kills the tag reference by the specified input mask.

Parameters:

Parameter: killPassword
Type - UInt32
Description - This parameter specifies the 32 bit kill password that must have been programmed into the tag prior to making this call.

Parameter: maskBank, maskAddress, maskLength, mask
Type - Byte, UInt16, UInt16, Byte[]
Description - These parameters are used to select a specific tag (or tags) to be killed. A description of these parameters an be found in the corresponding mask data section of the ReadTagData method above.

Parameter: errorCode
Type - out Byte
Description - This returns the underlying error code detailing any problems. Typical error codes are: 1 - the specified parameters are not supported, 2 - the password specified was wrong or provided insufficient privileges (e.g., 0 when associated parts of the tag are locked, and 3 - mask or data locations and/or lengths were not specified correctly. These error codes correspond to the values documented in Annex I of the EPCGlobal Gen2 Specification (Nov-2013, Version 2.0).

Return Value:

This function will return RFIDStatus.OK if it is able to kill the specified tag. It will return RFIDStatus.FAILED otherwise and return the extended error code in the errorCode parameter as documented in the ReadTagData/ErrorCode parameter description above.
Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Error Reporting

_____

RFIDStatus GetErrorList(ref UInt32 NumberOfErrors, ref UInt32[] When,
ref Uint16[] Code, ref Int16[] DByte1, ref Int16[] DByte2)

Function:
This method returns the list of errors detected and recorded by the sensor.
Parameters:

Parameter: NumberOfErrors
Type - ref UInt32
Description - Output value returning the number of errors detected by the sensor.

Parameter: When
Type - ref UInt32[]
Description - An array of timestamps, in seconds, for when the error were detected relative when the sensor was started (uptime).

Parameter: Code
Type - ref UInt16[]
Description - Output parameter returning an array of error codes for the detected errors.

Parameter: DByte1, DByte2
Type - ref Byte[], ref Byte[]
Description - These two arrays return two additional bytes of data relevant to the particular type of error that has been recorded.

Return Value:
This function will return RFIDStatus.OK if it is able to read and return the error list. It will return RFIDStatus.FAILED otherwise.
Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.
Additional Information:

{When[i], Code[i], DByte1[i], DByte2[i]} form the error information for the nth reported error.

The specific error codes are documented in the "*SensArray Communications Protocol, Part I*" document.

_____

RFIDStatus ClearErrorList()

Function:

This command clears the internal error list held by the sensor.

Return Value:

This function will return RFIDStatus.OK if it is able to clear the sensor's error list. It will return RFIDStatus.FAILED otherwise.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the sensor cannot be established or the module doesn't respond.

Additional Information:

The error list accumulates from the time the sensor starts or from the time the list is cleared. To conserve space, up to 32 errors will be recorded. Once the list is full, it overwrites the oldest error keeping the most recent 32 errors.

Appendix C

Low-Level Communication API Interface Guide:
RFID.ReaderComm Namespace

_____

public class RFIDReaderComm

<u>Function:</u>

This class encapsulates the low-level communications interface with the reader. In addition to providing setup, resetting, and teardown of the TCP socket used to communicate with the reader, it also packages up the binary protocol that is used in that communication.

<u>Additional Information:</u>

This class implements an on-demand approach to opening the TCP socket to establish communication the reader. Consequently, you will not find explicit Connect() or Disconnect() methods. You simply set up the IP address and port number for the socket, then call any of the SendMessage() methods. If the socket has never been opened, or was closed (due to a lost link, for example) the socket will be reopened. Once the application has finished communicating with the reader, call Shutdown(), which closes the socket and dispose of the object.

It is possible, however, to reuse this class by simply reinitializing it with a new set of IP address/port number values. If these are different from what was previously set up, the existing connection is closed, and a new one is automatically reestablished using these new values.

This class packages up the various commands sent to the reader to simplify the work that needs to be done at the application level. This packaging includes providing the correct set of header and trailer bytes, providing the correct message length, inserting the command code and data bytes, and finally calculating the checksum for the command. This packaged byte sequence is then sent to the reader and waits for a response.

The received data is also processed. The header and trailer bytes are verified, the response code is matched against the command code and the checksum is validated before the data length is and the data byte stream are extracted and returned to the caller. If any of these checks fails an Exception is thrown.

One additional comment in order. The RFID reader hardware is a standalone module embedded in the reader. Consequently, there are two primary subsystems within the reader that process commands. One handles the networking and top-level device management commands. The second handles the RFID protocol and reader type commands. At the binary protocol level, these commands are distinguished by having different header bytes. (Other than a different pair of header bytes, the command structures are identical.) The commands sent by the methods of this class need to be able to distinguish between these two

subsystems. This is done through the ReaderSubsystem enumeration. This enumeration has two values:

- ReaderSubsystem.RFID_Controller for commands targeted to the high-level controller (header bytes 0xB9/0x9B),

- And, ReaderSubsystem.RFID_Module for commands handled by the internal reader module (header bytes 0xA5/0x5A).

Specific details of the commands that are handled by the high-level controller are documented in "*SensArray Communications Protocol, Part I*". The specifics of the RF module's binary protocol are documented in "*SensArray Communications Protocol, Part II*".

Communication Timeout Member Variables

Function:

The following variables can be used to set and get the timeout values (in milliseconds) for the various phases of the communication process. The defaults for some of these may be a bit long, and need to be tuned, but to date have been chosen to minimize timeouts in our test lab. You will need to experiment with these values to determine whether shorter values will work reliably in your network, or if you are getting too many timeouts, you may find you need to increase these values a bit.

- Int32 ConnectionTimeout (get/set) - Default=15000 (15 sec.)

- Int32 DisconnectionTimeout (get/set) - Default=15000 (15 sec.)

- Int32 SendTimeout (get/set) - Default=5000 (5 sec.)

- Int32 ReceiveTimeout (get/set) - Default=15000 (15 sec.)

- Int32 ReceiveByteTimeout (get/set) - Default=15000 (15 sec.)

_____

ctor RFIDReaderComm(String IPv4Address, UInt16 PortNumber)
Function:

Create an instance of the RFIDReaderComm class initialized with the IPv4 address and port number passed in.

Parameters:

Parameter: IPv4Address
Type - String
Description - Input parameter specifying the IPv4 address that will be used to open the socket for communicating with the reader.

Parameter: PortNumber
Type - UInt16

Description - Input parameter specifying the TCP port number used when opening the socket for communicating with the reader.

Return Value:
On a successful return this will return an instance of the RFIDReaderComm class.
Exception(s) thrown:
A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.

_____

void Dispose()
Function:
The Dispose() method should be called prior to releasing any instance of the RFIDReaderComm class so that any memory allocated as part of opening the underlying socket resources can be freed properly.
Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the reader cannot be established or the module doesn't respond.

IP Settings and Socket Setup and Teardown

_____

void SetIPv4CommLink()
Function:
This method is used after the class has been created to change the IP address or the port number that the class will use to set up the TCP socket.
Parameters:
   Parameter: IPv4Address
      Type - String
      Description - Input parameter specifying the new IP address to assign to this communication channel. This must be in the standard dotted notation, e.g., 192.168.1.150.

   Parameter: PortNumber
      Type - UInt16
      Description - Input parameter specifying the new TCP port number to be used when opening the socket to communicate with the reader.

Return Value:
This method does not return a value.
Exception(s) thrown:
A System.FormatException will be thrown if the IP Address string is not a valid dot notation IP address.
Additional Information:

One side-effect of calling this method will be to close the existing socket and setup to reopen the socket if either the IP address or the port number change from what is currently set.

_____

void Shutdown()

Function:

Close and dispose of the underlying TCP socket.

Exception(s) thrown:

A System.TimeoutException will be thrown if the socket disconnect process doesn't complete within the disconnect timeout period.

Methods for Sending Messages and Receiving Replies

_____

Int32 SendMessage(ref Byte[] ResponseBuffer, Int32 MaxResponseLength,
*　　　　　ReaderSubsystem ModuleType, Byte MessageId,*
*　　　　　Byte[] MessageData = null, Int32 MessageDataLength = 0)*

Function:

Send a message to the reader expecting to receive a response. The ResponseBuffer parameter will contain any data content returned from the reader. The specific message ids and the data content of the sent messages are received replies are documented in the Additional Information section of the top-level class description above.

Parameters:

　　Parameter: ResponseBuffer

　　　　Type - ref Byte[]

　　　　Description - Output parameter providing a buffer for the response data in the reply to the message to be returned to the calling code.

　　Parameter: MaxResponseLength

　　　　Type - Int32

　　　　Description - Input specifying the maximum number of bytes to be returned in the reply. This should be less than or equal to the number of elements in the ResponseBuffer array.

　　Parameter: ModuleType

　　　　Type - ReaderSubsystem

　　　　Description - Input parameter which specifies the subsystem that will be handling the command. The ReaderSubsystem usage is described in the Additional Information section of the class overview.

　　Parameter: MessageId

　　　　Type - Byte

Description - Input parameter which specifies the command that is to be sent to the reader.

Parameter: Parameter Name
    Type - String
    Description - Description

Parameter: MessageData
    Type - Byte[]
    Description - Optional Input array comprising the data content of the message to be sent to the reader. If there is no data content in the outgoing message, this can be eliminated from the method call.

Parameter: MessageDataLength
    Type - Int32
    Description - Optional Input specifying the number of bytes in the outgoing message data. If there is no data content to the message, this parameter can be eliminated from the call.

Return Value:
Int32 value is returned with the length of the data content of the response.
Exception(s) thrown:
A System.TimeoutException will be thrown if the connection to the reader cannot be established or the module doesn't respond.
Additional Information:
The actual binary data stream that is sent to the reader includes header and trailer bytes, the message length and a checksum along with the command code and data. Similarly, on return, the response has a similar structure. This method packages this information appropriately and extracts the returned data removing the need to perform these operations from the application level code.

_____

void SendMessageNoReply(ReaderSubsystem ModuleType, Byte MessageId,
                    Byte[] MessageData = null, Int32 MessageDataLength = 0)

Function:
Send a message to the reader without expecting a reply.
Parameters:
See the Parameters section of the SendMessage() method above for the description of the parameters to this method.
Return Value:
There is no return value from this method since we are sending the message expecting the reply to be obtained using the DoReceive() method.
Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the module doesn't respond.

Additional Information:

There are only a handful of commands where this method is appropriate. These include rebooting the device and starting a continuous read cycle.

_____

Int32 DoReceive(ref Byte[] ResponseBuffer, Int32 MaxResponseLength,
ReaderSubsystem ModuleType, Byte MessageId, Int16 AltMessageId = -1)

Function:

Wait for formatted binary protocol data replies to be sent from the reader. Return the resulting data when a complete message reply has been received.

Parameters:

A description of the output parameter ResponseBuffer, and input parameters MaxResponseLength and ModuleType can be found in the Parameters section of the SendMessage() method above.

    Parameter: MessageId
        Type - Byte
        Description - Input parameter passing in the id corresponding to the outgoing message that corresponds to the expected reply.

    Parameter: AltMessageId
        Type - Int16
        Description - Optional Input parameter passing in the id corresponding to a second message type that might occur while receiving data. If this parameter is left out of call, this method will only return messages corresponding to the primary id set by the MessageId parameter. Other responses will throw an exception. The primary use of this parameter is described in the Additional Information section below.

Return Value:

Int32 value is returned with the length of the data content of the response.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or the module doesn't respond.

Additional Information:

The primary use for this method is to listen for tag data replies during continuous tag inventory read cycles. The read cycle is initiated by issuing a Start Continuous Inventory command. Tag data is read by receiving tag data in a continuous loop by calling DoReceive(). When the application is ready to stop the continuous inventory cycle, a Stop Continuous Inventory command is issued while continuing to monitor responses using DoReceive() with the primary MessageId being the one for tag reads and the alternative message id (AltMessageId) being the one for the Stop Continuous Inventory command.

Note that in this situation, both the Start Continuous Inventory command and the Stop Continuous Inventory command would be sent using the SendMessageNoReply() method.

_____

Int32 DoReceiveByte(ref Byte[] ResponseBuffer, Int32 MaxResponseLength)

Function:

Read a single byte from the incoming data stream.

Parameters:

A description of the output parameter ResponseBuffer, and input parameter MaxResponseLength SendMessage() method above.

Return Value:

Int32 value is returned with the length of the data content of the response.

Exception(s) thrown:

A System.TimeoutException will be thrown if the connection to the reader cannot be established or if no data comes in on the socket in the designated timeout period.

Additional Information:

There are two primary purposes for this command. One is to clear the data input buffer if something glitches in the communication channel. The second is to provide direct access to the full data stream for debugging purposes.

Appendix D

Administrative Notification Listener Class API Interface Guide: RFID.Notifications Namespace

_____

public class SysMessageListener

<u>Function:</u>

This class is used to launch a TCP listener on the IP Address of a specific interface and designated port. The listener waits for connections, reads a tag data stream from the connecting program and passes the tag data messages through the registered callback to the application for processing.

OVERIVEW:

The SysMessageListener is a lightweight class that manages binary UDP Notification messages from SensThys readers. Readers broadcast status messages on port 3984. Applications can listen for these messages using this class.

Applications can listen for these messages and receive events corresponding to changes in reader state. (e.g., 'started')

This model, since it is connectionless, has the benefit of scalability and does not require a central server to maintain multiple open connections.

USING THE CLASS:

Briefly, one creates an instance of the class, specifying a listening port upon which to wait for UDP broadcast messages.

At this point, delegates may be assigned to events supported by the class. This is currently limited to system boot events but future extensions may include GPIO events, error notifications etc.

Note that the calling application needs to spawn a thread for the StartListening() call to run within since StartListening() will block listening for incoming messages. Otherwise other events cannot be processed including the registered RebootMessage handler.

Once the listener has begun listening, all interactions with the class are through the pre-assigned event handlers.

Events are of the form:

```
public class SensThysListenerEventArgs : EventArgs
{
        public string ClientIPAddress { get; set; }
```

```
        public DateTime EventTime { get; set; }
        public int ErrorType { get; set; }
        public string StrData { get; set; }
}
```

Where:

- ClientIPAddress is the address of the connecting reader.

- EventTime is when the event occured.

- ErrorType returns an integer representing the state of the system. (0=No error)

- StrData is any data payload associated with the event.

_____

SysMessageListener(int Port = 3984)

Function:

Constructor for the SysMessageListener class

Parameters:

Parameter: Port

Type - int

Description - UDP Port number on which this class listens for system messages from the reader. Leaving this parameter out will set this interface up to utilize the default UDP port number 3984.

Return Value:

Call to this constructor returns an initialized SysMessageListener object

Additional Information:

Currently, the SensArray readers do not support configuring the UDP port over which system notification messages are delivered.

_____

public event RebootEventHandler ReaderRebooted;
Function:

This is the interface used to register the event handler for when the reader reports reboot events. Your event handler should have a signature that matches the RebootEvenHandler delegate interface shown below.

Your event handler should be registered as follows:

notificationListenerInstance.ReaderRebooted += myRebootHandler;

public delegate void RebootEventHandler(Object sender, SensThysListenerEventArgs e);
_____

public bool listening;

Function:

This property can be queried to determine whether the code is currently listening for event messages from the reader.

_____

StartListening(void)

Function:

This method starts the listening and data callback process. StartListening() should be called when your application is set up and is ready to accept and respond to system notifications.

Parameters:

None

Return Value:

No value is returned

Exception(s) thrown:

A ListenerCallbackNotSet exception is raised if this method is called prior to the ReaderRebooted event handler being registered.

An AlreadyListening exception will be thrown if StartListening() is called more than once for the associated SysMessageListener instance without calling StopListening(). If you are unsure of the state, you can always call StopListening() independent of the current state of the listener, or you can check the listening property.

Both of these exceptions are indicative of a code programming logic issue. However, if you want to catch and handle these error explicitly in your code, the best way to do this is to derive your own class from SysMessageListener and create a method to wrap StartListening() in a try/catch block. This method would then be invoked from your thread setup call.

Additional Information:

This method starts listening for incoming UDP messages and invokes the event handler that has been associated with the specified message. Note that this method currently blocks waiting to receive messages, so it should be run within a thread if your application needs to do other processing while listening for messages.

StopListening(void)

<u>Function:</u>

Terminates the listening service.

<u>Parameters:</u>

None

<u>Return Value:</u>

No value is returned

<u>Exception(s) thrown:</u>

None

_____

Example Program

```csharp
using System;
using System.Threading;
using RFID.Notifications;

namespace SystemMessageApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create an instance of the Listener using the default port (3984)
            SysMessageListener sysMsgListener = new SysMessageListener();

            // Add appropriate delegates / event hooks
            sysMsgListener.ReaderRebooted += c_ReaderBooted;

            Console.WriteLine("Starting Listener...");

            new Thread(new ThreadStart(sysMsgListener.StartListening)).Start();

            Console.WriteLine("Listener Active.");

            while (true)
            {
                // Perform other work in Main here or run a GUI
                Thread.Sleep(1000);
                Console.WriteLine("Tick...");
            }

            //Main done.
        }

        //*************** Event Handlers ***************

        // A tagList payload has arrived at the Listener.
        static void c_ReaderBooted(Object sender, SensThysListenerEventArgs e)
```

```csharp
        {
            string strTagList = e.StrData;

            if (strTagList.Length > 0)
            {
                Console.WriteLine("********************************************************");
                Console.WriteLine("Reboot message Received from: {0}", e.ClientIPAddress);
                Console.Write(e.StrData);
                Console.WriteLine();
            }
            else
            {
                Console.WriteLine("No data in MessageReceived from {0}.", e.ClientIPAddress);
            }
        }
    }
}
```

Appendix E

Unsupported/Unknown Commands

Some members of the SensThys reader family support capabilities that other family members don't.

The API methods that are not supported on different platforms can certainly be called from your application code. However, if the method is not supported it will throw one of two possible exceptions:

- ReaderReceivedNotImplementedResponse
- ReaderReceivedNoHardwareSupportResponse

In addition to the usual base class Exception fields (such as message) these add a command code field that can be used to identify the specific underlying code corresponding to the API method that was called.

Note that if your application needs to support different platforms, you can wrap the functions that call the hardware specific methods in try/catch blocks, or you can always query the reader type and make the call only when the hardware support is available.