A close-up of numbers

Description automatically generated





**JSR .NET SDK: User's Guide**

**Revision: 1.2.14**

**Date: 11/11/2024**

[This text is hidden and will not print]

To update the document title, revision number and revision date, press CTRL-A to select all text, then press F9 and enter the values in the prompts. Then view the Print Preview (CTRL-ALT-I) to update all fields.

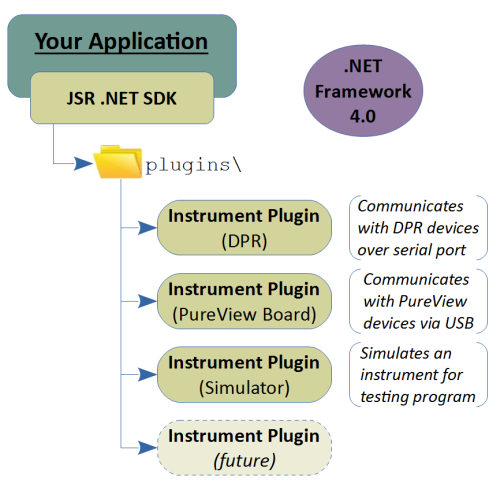
DocTitle Field [JSR .NET SDK: User's Guide]

RevNum Field [1.0.1]

RevDate Field [5/24/2019]

## Introduction

The JSR .NET SDK is a framework designed to assist with creating custom applications that use the JSR Ultrasonics Pulser-Receiver Instruments. This framework consists of two types of libraries that work together to allow your application to control the Instrument.

**SDK library**

This is a single DLL assembly which provides the interfaces and classes for controlling an Instrument. The primary class of the SDK is the JSRDotNETManager that handles the primary operations:

* Loading Instrument Plugins
* Discovering attached Instruments and Pulser-Receivers
* Setting and querying the Pulser-Receiver configuration

**Instrument Plugin Libraries**

This is a set of DLL assemblies that are dynamically loaded at run time by the SDK and provide the specific implementations for communicating directly with the Instruments. Since these plugins are loaded dynamically, they are designed to be updated or added to without requiring a recompile of your application. For example, if you need to access a new family of Pulser-Receiver Instruments, then you can simply include the plugin DLL file in your plugins\ folder and access to that Instrument type will be available the next time your application is run.

NOTE: To ensure compatibility of the Instrument Plugins and SDK libraries, the major revision number of the Instrument Plugin and SDK library assemblies must match. If an upgrade is required that updates the major revision number of the Instrument Plugins, then your application will need to be recompiled with the latest version of the SDK libraries.

The purpose of this document is to guide you through the process of integrating the JSR .NET SDK into a working application. The full example source for the application can be found in the JSRDotNETSDK\_Sample project and a copy of the source is included in this document.

## Getting Started With the JSRDotNETSDK

The purpose of this guide is to get you up and running quickly using the JSRDotNETManager to communicate with a Pulser-Receiver Instrument. There is also an advanced section that goes into more detail regarding configuring the Instrument Plugins to set specific options when working with the Instruments.

The following steps are required to get started with the SDK in your project:

1. Configure your project to use the JSR .NET SDK.
2. Create a JSRDotNETManager instance and load the Instrument Plugins.
3. Select and configure the Instrument Plugins to use.
4. Start the Instrument & Pulser-Receiver discovery.
5. Select the desired Pulser-Receiver and control the unit.
6. Shut down the JSRDotNETManager during Application Exit.

## Requirements

The JSR .NET SDK was created using Visual Studio 2010 Professional and the .NET Framework v4.0.

The example projects included with the SDK have been successfully built and run using the following development environments:

* Visual Studio 2010 Professional
* Visual Studio 2010 Express
* Visual Studio 2012 Express
* Visual Studio 2013 Community
* Visual Studio 2015 Community
* Visual Studio 2017 Community
* Visual Studio 2017 Professional
* Visual Studio 2019 Community
* Visual Studio 2019 Professional
* Xamarin Studio 5.10.3 with the Windows 7/.NET Framework 4 SDK installed

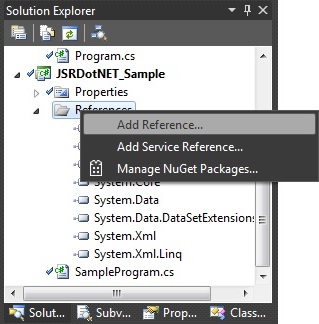
## Configure Your Project to Use the JSR .NET SDK

To use the JSR .NET SDK, your project needs to reference one assembly which can be found in the Libs/ directory of the JSRDotNETSDK\_Deliverable distribution archive:

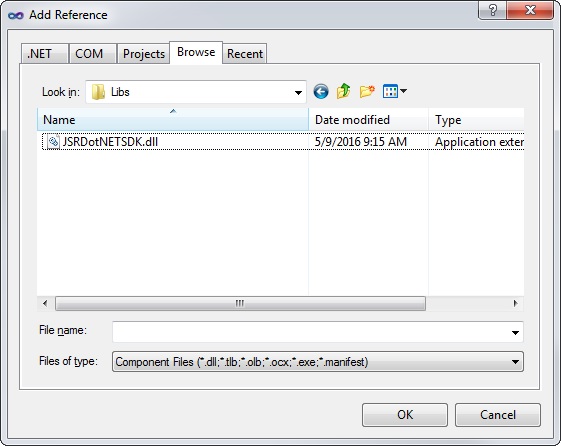
* JSRDotNETSDK – this component contains the interface and class definitions that are common to all plugin modules.

NOTE: By adding a reference to this assembly, your project should (and may need to) be recompiled if this library component is updated (e.g. using a new version).

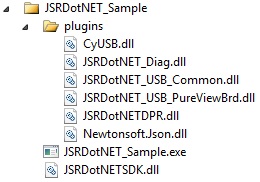
To add these references to your project, expand your project in the Solution Explorer window. Right-click on the “References” entry and select “Add Reference…”



In the “Add Reference” window, click on the “Browse” tab and then browse to the location of the Assemblies to be referenced. Select the Assembly DLL file to be added as reference for your project. Press “OK” to create the reference to the Assemblies. Once the module is added to your project, it will be included in the “References” list for your project.



## Create a JSRDotNETManager Instance and Load the Instrument Plugins.

The next step in your application is to determine where you are going to store the Instrument Plugins and then set up your application to use the JSRDotNETManager to load the plugins from this location. The specific location where these plugin files are stored is not important, but your application will need to pass this path to the JSRDotNETManager before loading the plugins. If you do not provide a path for the plugins directory, the SDK will use the default Instrument Plugin location of the plugins/ subdirectory underneath where your application is running. An example of the default path is shown in the picture on the right.

If you are using your own directory for the plugins, you can use the System.Reflection library to retrieve the current location of your application and build the plugin location from there. The following example shows how the SDK creates its default location for the plugins directory.

**NOTE:** Only one instance of the JSRDotNETManager should be created by your application. The behavior of the Instrument Plugins is undefined if there is more than one instance of the JSRDotNETManager.

**using** System**.**Reflection**;**

**···**

// We need the path to the plugins directory that contains the JSRDotNET

// plugins. In this case, the plugins are in a subdirectory where the

// executable is located. Use the Assembly functions to get the application

// directory.

string pluginPath **=** System.IO**.**Path**.**GetDirectoryName**(**

Assembly**.**GetExecutingAssembly**().**Location**)** **+** @"\plugins"**;**

This path is either provided to the JSRDotNETManager during construction or by setting the PluginPath property in the JSRDotNETManager instance. When you are ready to detect the plugins, call the LoadPlugins() method on your JSRDotNETManager instance. If there is an error with your path or with loading the plugins, an exception will be thrown.

**using** JSRDotNETSDK**;**

**···**

// Create a new JSRDotNETManager and provide the path to the plugins

JSRDotNETManager jsrManager **=** **new** JSRDotNETManager**(**pluginPath**);**

// OR

// Create a new JSRDotNETManager using the default constructor and set the

// plugin path using the PluginPath property

JSRDotNETManager jsrManager **=** **new** JSRDotNETManager**();**

jsrManager**.**PluginPath **=** pluginPath

// Load all of the Plugin dlls - this may throw an exception

// if there is an error with the path.

jsrManager**.**LoadPlugins**();**

## Select and Configure the Instrument Plugins to Use

If the plugins were loaded successfully, then the JSRDotNETManager instance has a list of the plugins that were found in the directory. The following functions in the JSRDotNETManager provide information about the loaded plugins:

* GetPluginNames()

Returns a List<string> containing the name of each plugin discovered in the plugin directory. Throws an ExceptionJSRDotNET if the plugins are not loaded.

* GetPluginLibraryMetadata(string strPluginName)

Returns a Metadata object that contains information about the named plugin. Throws an ExceptionJSRDotNET if the plugins are not loaded or the name is not found.

* GetPluginLibraryInstance(string strPluginName)

Returns a reference to the plugin as an IJSRDotNET instance. Throws an ExceptionJSRDotNET if the plugins are not loaded or the name is not found.

The following example code uses these functions to display information about all of the plugins found.

// Get the list of plugins found (these are just the names)

**foreach** **(**string name **in** jsrManager**.**GetPluginNames**())**

**{**

Console**.**WriteLine**(**"Found plugin: '" **+** name **+** "'"**);**

// Get the metadata (info) about the plugin

IJSRDotNETLibMetadata pluginInfo **=** jsrManager**.**GetPluginLibraryMetadata**(**name**);**

// Print some info about the plugin

Console**.**WriteLine**(**" - " **+** pluginInfo**.**FriendlyName **+** " - " **+**

string**.**Join**(**", "**,** pluginInfo**.**SupportedModels**)** **+**

" (" **+** string**.**Join**(**", "**,** pluginInfo**.**ConnectionType**)** **+** ")"**);**

**}**

Example output:

Found plugin: 'JSRDotNET\_DPR'

- DPR Series - DPR300, DPR500 (SERIALPORT)

Found plugin: 'JSRDotNET\_DiagLib'

- Generic Simulator - GenericInstrument (SOFTWARE)

Once you have determined which Instrument Plugin will be used with your application, you will need to add the name of that plugin to the managed plugins list in the JSRDotNETManager. All of the plugins in this list will be used for Pulser-Receiver discovery and all events from these plugins will be marshalled through the JSRDotNETManager event handler. Your application can also configure the Instrument Open Options for each plugin in this list using the JSRDotNETManager (more on this in a later section of this document).

The following example code adds all the discovered Instrument Plugins to the managed list. Your application can decide which plugins it wants to add to the managed list (or it can add all of them).

// Add all of the plugins to the managed list in the JSRDotNETManager

**foreach** **(**string name **in** jsrManager**.**GetPluginNames**())**

**{**

jsrManager**.**AddManagedPlugin**(**name**);**

**}**

## Start the Instrument & Pulser-Receiver Discovery

Once the Instrument Plugins have been added to the managed list in the JSRDotNETManager, your application can begin to look for any Instruments and Pulser-Receivers that are attached to the PC. Each Instrument Plugins specifies which interface(s) that it uses to look for devices (e.g. USB, SERIAL, etc.). This can be used to help determine which plugin your application wants to use.

The primary event reported by the JSRDotNETManager is the NotifyEventHandler that sends different types of events depending on what is happening in the system. The following is the list of events reported:

* *DISCOVERY\_STARTED*

This event occurs when the Pulser-Receiver discovery within a single Instrument Plugin has been started.

* *DISCOVERY\_COMPLETE*

This event indicates that the initial Pulser-Receiver discovery by an Instrument Plugin is completed. If Discovery remains enabled and the Instrument Plugin supports dynamic discovery (e.g. USB), it is possible for new devices to be reported after this event occurs.

* *PULSER\_RCVR\_DISCOVERED*

This event occurs when a new Pulser-Receiver has been discovered. The EventArgsManagerNotify object contains a reference to the IPulserReceiverIdentity for the specific Pulser-Receiver. This object also includes an array of version information strings and the maximum frequency for this device.

* *PULSER\_RCVR\_DETACH*

This event occurs when an existing Pulser-Receiver has been detached. The EventArgsManagerNotify object contains a reference to the IPulserReceiverIdentity for the specific Pulser-Receiver that was removed. When this event occurs, the application should release any references to this Pulser-Receiver so the resources can be freed. If the detached device was the same device that was selected in the JSRDotNETManager, then the WasSelected property of the EventArgsManagerNotify object will be set to true.

* *CURRENT\_PULSER\_RCVR\_CHANGED*

This event occurs when the selected Pulser-Receiver in the JSRDotNETManager has been changed. The manager can select only one Pulser-Receiver device that it can control. This event indicates that a new Pulser-Receiver device has been selected and the EventArgsManagerNotify object contains a reference to the IPulserReceiverIdentity of the new device.

* *PROPERTY\_CHANGE*

This event occurs when a property has been changed in the instrument. This can be the result of a manually configured item or something that has changed in the system asynchronously.

* *ERROR*

This event occurs when an error occurs during an operation. Additional information can be retrieved via the ErrorMsg, ErrorText and ExceptionTypeInfo properties of the EventArgsManagerNotify object.

Since this section is mainly concerned with discovering the attached Pulser-Receiver devices, the application will need to add an event function that can handle the Notify events from the JSRDotNETManager. The first step is to create a function that will receive events from the JSRDotNETManager. The following is a sample function that prints the name of the event that occurred based on the value of the NotifyType property in the EventArgsManagerNotify instance. The event handler method is registered with the JSRDotNETManager instance by adding it to the NotifyEventHandler event handler.

// Event handler for notify events from the JSRDotNETManager

**internal** void OnJsrManagerNotify**(object** sender**,** EventArgsManagerNotify e**)**

**{**

Console**.**WriteLine**(**"OnJsrManagerNotify - event: " **+** e**.**NotifyType**.**ToString**());**

**}**

**···**

// Add our NotifyEventHander method to the jsrManager

jsrManager**.**NotifyEventHandler **+=** **this.**OnJsrManagerNotify**;**

In the following example, the function has been expanded to print the type of event that occurred and information about the Pulser-Receiver. An AutoResetEvent m\_WaitEvent was added to allow this function to exit out of the discovery wait time prematurely if no devices were found or an error occurs. This example doesn’t care about the CURRENT\_PULSER\_RCVR\_CHANGED event, so the function returns early if this type of event occurs.

**internal** void OnJsrManagerNotify**(object** sender**,** EventArgsManagerNotify e**)**

**{**

// Print information common to all events

Console**.**WriteLine**();**

Console**.**WriteLine**(**"EVENT{0}: {1}"**,**

**(**string**.**IsNullOrWhiteSpace**(**e**.**PluginName**)** **?** "" **:** "[" **+** e**.**PluginName **+** "]"**),**

e**.**NotifyType**);**

**if** **(**e**.**PulserReceiverId **!=** **null)**

**{**

Console**.**WriteLine**(**"DEVICE: {0}"**,** e**.**PulserReceiverId**.**ToString**());**

**}**

// Print additional info for each event type

**if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**PULSER\_RCVR\_DETACH**)**

**{**

**if** **(**e**.**WasSelected**)**

Console**.**WriteLine**(**"DETACHED: Device was the selected device"**);**

**}**

**else** **if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**DISCOVERY\_COMPLETE**)**

**{**

m\_WaitEvent**.**Set**();**

**}**

**else** **if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**ERROR**)**

**{**

**if** **(**0 **==** **(**e**.**DiscoverState **&** DiscoveryStateFlags**.**PENDING**))**

**{**

Console**.**WriteLine**(**" ERROR MSG: " **+** e**.**ErrorMsg**);**

Console**.**WriteLine**(**"ERROR TEXT: " **+** e**.**ErrorText**);**

m\_bErrorOccurred **=** **true;**

m\_WaitEvent**.**Set**();**

**}**

**else**

**{**

Console**.**WriteLine**();**

Console**.**WriteLine**(**e**.**ErrorMsg**);**

Console**.**WriteLine**();**

**}**

**}**

**}**

Now that the Event Handler has been properly set up to save the detected Pulser-Receiver devices, the Instrument discovery can begin. To start this process, call the SetDiscoveryEnable(true) method which will set the InstrumentDiscoveryEnable property to true for all of the Instrument Plugins added to the managed list. Depending on the needs of your application, it can either wait for a pre-set amount of time or until a specific set of devices have been discovered. The easiest implementation is to wait around 30 seconds for device discovery to allow the SERIAL devices to be discovered.

**using** System**.**Threading**;**

**···**

// Start the device discovery

jsrManager**.**SetDiscoveryEnable**(true);**

// Sleep for 20 seconds to allow the devices to be discovered

Thread**.**Sleep**(**20000**);**

NOTE: The InstrumentDiscoveryEnable behavior will vary slightly depending on the interface that is used to communicate with the Instrument. For most cases, your application can enable Instrument Discovery and leave it enabled. The behaviors are as follows:

USB

The driver will report any new USB Instruments while Instrument Discovery is enabled.

SERIAL

The driver will scan all the COM Ports one time when Instrument Discovery is enabled. To re-scan the COM Ports, Instrument Discovery needs to be disabled and then enabled.

SOFTWARE

These are simulated devices that will be reported the first time Discovery is enabled.

NOTE: It is possible for the SERIAL driver to interfere with non-pulser/receiver instruments that also use serial ports.

The SERIAL driver will probe each serial interface that it finds for the presence of compatible pulser/receiver instruments. This behavior can have unintended consequences if a serial interface has another type of device that may consider the pulser inquiry as a malformed command.

Controlling the startup of instrument discovery relative to the startup of other serial devices may be necessary.

A more advanced discovery routine could be used that allows a user to break out of the device discovery wait by pressing a key or to stop the wait if an error occurs or the plugin reports no devices found. Here is an example:

Console**.**WriteLine**(**"Waiting for devices... Press any key to stop"**);**

Console**.**WriteLine**();**

// Wait for 20 seconds (in milliseconds)

int sleepWait **=** 20000**;**

**while** **(**sleepWait **>** 0**)**

**{**

sleepWait **-=** 100**;**

// Exit wait if an error occurs, no devices were found or

// the user requested a stop.

**if** **((**m\_WaitEvent**.**WaitOne**(**100**))** **||** **(**Console**.**KeyAvailable**))**

sleepWait **=** 0**;**

**}**

// Flush any kepresses in the buffer

**while** **(**Console**.**KeyAvailable**)**

Console**.**ReadKey**(true);**

## Select the Desired Pulser-Receiver and Control the Unit

You can use the GetPulserReceivers() method in the JSRDotNETManager to retrieve the list of Pulser-Receiver devices that was found during discovery. This will be an array of IPulserReceiverIdentity objects that contains information about the Instrument and Pulser-Receiver. This can also be used to select the current Pulser-Receiver that the JSRDotNETManager will control using the SetCurrentPulserReceiver() method. Once a Pulser-Receiver has been selected, the JSRDotNETManager provides the complete list of functions found in the IPulserReceiver interface for getting information about and controlling a specific Pulser-Receiver device. The following example code reads a couple of properties and prints the information for each of the Pulser-Receiver devices that were found.

// Get the list of the discovered Pulser-Receivers, select each one a print out

// out some parameters for each.

**foreach** **(**IPulserReceiverIdentity prID **in** jsrManager**.**GetPulserReceivers**())**

**{**

Console**.**WriteLine**(**" P/R: {0}[{1}] - IDX {2}"**,** prID**.**InstrumentId**.**ModelName**,**

prID**.**InstrumentId**.**SerialNum**,** prID**.**PulserReceiverIndex**);**

**try**

**{**

// Set this Pulser-Receiver as the current device - this function will

// cause the CURRENT\_PULSER\_RCVR\_CHANGED event to be triggered.

jsrManager**.**SetCurrentPulserReceiver**(**prID**);**

// We can now call all the functions available in the IPulserReceiver interface

// on the selected Pulser-Receiver by using the functions in JSRDotNETManager

// Some values are discrete values – int, bool, double:

Console**.**WriteLine**(**" BANDWIDTH: {0:F}"**,** jsrManager**.**MaxFrequency**);**

Console**.**WriteLine**(**" GAIN: {0:F2}"**,** jsrManager**.**Gain**);**

Console**.**WriteLine**(**" IS\_TRIGGERED: {0}"**,**

**(**jsrManager**.**TriggerEnable **?** "YES" **:** "NO"**));**

// Some values return an enum value:

Console**.**WriteLine**(**" SIGNAL\_SELECT: {0}"**,**

jsrManager**.**ReceiverMode**.**ToString**());**

// And some values return an indexed value:

Console**.**WriteLine**(**" LOW\_PASS: {0:F0}"**,**

jsrManager**.**LowPassFilterValues**[**jsrManager**.**LowPassFilterIndex**]);**

// Some values are not supported on all devices and have a ‘Supported’ property:

**if** **(**jsrManager**.**HVSupplyEnableSupported**)**

Console**.**WriteLine**(**" HV\_SUPPLY: {0:F0}"**,**

**(**jsrManager**.**HVSupplyEnable **?** "ENABLED" **:** "DISABLED"**));**

**}**

**catch** **(**Exception ex**)**

**{**

Console**.**WriteLine**(**"Exception occurred during processing: " **+** ex**.**Message**);**

**}**

Console**.**WriteLine**();**

**}**

Another method for determining which Pulser-Receiver to use is to retrieve the list of discovered Instruments by calling the GetInstruments() method in the JSRDotNETManager. By providing the IInstrumentIdentity instance for the desired Instrument into the GetPulserReceivers() method in the JSRDotNETManager, a (smaller) list of IPulserReceiverIdentity instances will be returned for the Pulser-Receiver devices owned by that Instrument.

## Shut down the JSRDotNETManager during Application Exit

During the exit of your application, the Shutdown() method on your JSRDotNETManager instance will need to be called. This ensures all the plugins are properly closed and any threads/resources will be released. Once this is called, your application will no longer be able to communicate with the Instruments and Pulser-Receivers. The application should also make sure Instrument Discovery is disabled and any references to Pulser-Receiver objects are released before calling the Shutdown() method.

// Make sure device discovery is turned off

jsrManager**.**SetDiscoveryEnable**(false);**

// Unregister our NotifyEventHander

jsrManager**.**NotifyEventHandler **-=** **this.**OnJsrManagerNotify**;**

// Make sure any local references to Pulser-Receivers have been freed...

// Application is finished with the JSRDotNETManager; call the

// Shutdown() function

jsrManager**.**Shutdown**();**

NOTE: If you are using the Instrument Plugin for the DPR devices, there may be a short pause (1-2 seconds) during shutdown of your application if this occurs during device discovery or while a serial transaction is in process.

# Example Projects

The JSR .NET SDK includes 5 sample programs that demonstrate the use of the SDK for displaying information from and controlling the Instruments and Pulser-Receivers attached to the Client PC. Each one of these programs has their own Visual Studio project and a solution containing all of the projects can be found at the top level directory. The projects were created using Visual Studio 2010 and have been tested to be compatible with Visual Studio versions through Visual Studio 2017.

* JSRDotNET\_ConsoleExample

This is a simple console application that prompts for a serial port and then displays the Info strings for the first Pulser-Receiver that is detected. It demonstrates the following:

* + Instrument Plugin loading
  + Instrument Plugin Open Options and Port Include configuration
  + Pulser-Receiver discovery and event handling
  + Retrieving and displaying information for a Pulser-Receiver
* JsrConsole

This is a full featured console application that works all of the loaded Instrument Plugins to detect all of the attached and simulated Pulser-Receiver devices. Each Pulser-Receiver can be selected individually and queried and configured. All of the settings are discovered dynamically depending on what the device supports. It demonstrates the following techniques:

* + Instrument Plugin loading
  + Instrument Plugin Open Options parsing and configuration
  + Pulser-Receiver discovery and event handling
  + Discovering the settings supported by a Pulser-Receiver using the list of PulserSettingInfo instances retrieved from that Pulser-Receiver
  + Displaying and configuring Pulser-Receiver settings using the PulserSettingInfo instances
* JSRDotNET\_FormExample
  + This example project provides a UI application that uses the JSRDotNETManager for controlling a Pulser-Receiver similar to the JSR Control Panel. A list box shows all the discovered devices and only one Pulser-Receiver can be viewed or controlled at a time. It demonstrates the following:
  + Instrument Plugin loading
  + Instrument Plugin Open Options parsing and configuration
  + Pulser-Receiver discovery and event handling
  + Selecting and using a single Pulser-Receiver instance by using the JSRDotNETManager
  + Automatic update of DPR500 device during Pulser hot swap.
* JSRDotNET\_MultiForm
  + This example project provides a demonstration of using the Pulser-Receiver Instances directly instead of using the JSRDotNETManager. This is an MDI application with a different Form for each Pulser-Receiver that controls that device independently. This project demonstrates the following techniques:
  + Instrument Plugin loading
  + Instrument Plugin Open Options parsing and configuration
  + Pulser-Receiver discovery and event handling
  + Retrieving an IPulserReceiver instance for each discovered Pulser-Receiver
  + Querying and Controlling Pulser-Receivers independently using the IPulserReceiver instances.
* JSRDotNET\_Sample

This example project provides the source code for the example in this document.

The following list describes the other components included in the JSRDotNET\_Examples directory:

* LEGAL.TXT

The legal statements for open source software included in the JSR .NET SDK and Instrument Plugins.

* LICENSE\_EXAMPLES.TXT

This file contains the software license for the Example files included in this directory.

* LICENSE\_SDK.TXT

This file contains the software license for the JSR .NET SDK and Instrument Plugins. By using the DLL assemblies provided with the JSR .NET SDK, you agree to be bound by this license.

The location of the JSR .NET SDK components can be found in the following directories under the JSRDotNETSDK\_Deliverable directory:

* Libs

This directory contains the primary JSR .NET SDK DLL assembly file.

* Plugins

This directory contains the Instrument Plugin Libraries. The SDK will need to be provided the location of this directory (or the plugin directory in your application) during initialization.

# Advanced Topics

## Configure the Plugin Open Options before Starting Instrument Discovery

Each plugin library has a set of open criteria, called the Instrument Open Options, which affects how that plugin will search for new Instruments and Pulser-Receivers. This consists of the following configuration items:

* **Model Number** – *Specifies the Instrument Model Number to detect/connect.* If a Model Number is set, then the plugin will ignore any Instruments whose Model Number does not match this value. To uniquely identify a specific Instrument, both the Model Number and Serial Number should be used.
* **Serial Number** – *Specifies the Instrument Serial Number to detect/connect.* If a Serial Number is configured, the plugin will ignore any Instruments whose Serial Numbers do not match this value.
* **Ports to Include** – *Specifies the ports to be used for detection.* Only ports (e.g. COM ports, IP Addresses, etc.) that match the items in this list will be used to search for Instruments. If this list is empty, then all available ports will be used.
* **Ports to Exclude** – *Specifies the ports to exclude during Instrument detection.* When the plugin is searching for instruments, it will not use the ports that match items in this list. A port cannot be listed in both the Include and Exclude lists. If this list is null/empty, then the ports listed in the Include list or all available ports (if the Include list is empty) will be used.
* **Allowed Interfaces** – *Specifies the connection types to be used for Instrument detection.* When the plugin performs its search, it will only use the connection types listed in the allowed list or all connection types if this list is null/empty. If this list is not empty and a plugin’s interface is not listed, then it will not be able to detect any Instruments.
* **Plugin-Specific Options** – *Specifies the plugin-specific options for the Instruments.* Each plugin may have an additional set of options that controls the operation of that plugin. The list of plugin-specific open options can be retrieved from the info object received by calling the GetPluginLibraryMetadata() method. The open options is provided as an array of strings with the following formats:
  + **ENUM Open Option**:

<Open Option Name>:<option1>,<option2>...

* + **INT Open Option**:

<Open Option Name>:int|<min>|<max>

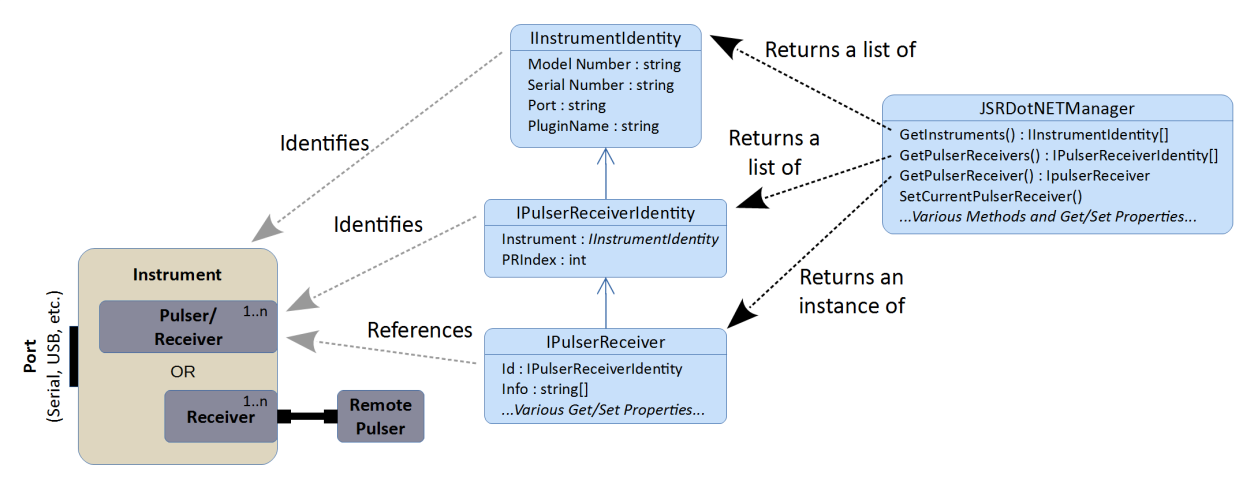
When setting the Open Option, your application provides the open option name and the value (as a string) that it wants to use. For numeric Open Options, the number is provided as a string.

In order to configure these options, your application configures the InstrumentOpenCriteria object that is retrieved from that plugin using the InstrumentOpenCriteria property *before* starting Instrument Discovery. The JSRDotNETManager can also be used to configure some of these options, such as the Plugin-Specific Options.

## JSR .NET SDK Architecture

The following diagram illustrates the relationship between the Interfaces and Classes in the JSR .NET SDK. There are two different architectures for the Pulser-Receiver hardware:

1. *Combined Pulser-Receiver* – The Pulser and Receiver are in the same unit (such as that found in the DPR300 and the USB Pulser-Receivers).
2. *Remote Pulser* – The Receiver is contained in the Instrument and the Pulser is an interchangeable remote unit (this is the architecture of the DPR500).



An Instrument is represented by an IInstrumentIdentity instance that contains the Model Number, Serial Number and Port information for that device. This object can be used to retrieve the list of Pulser-Receivers owned by this Instrument from the JSRDotNETManager.

An individual Pulser-Receiver is identified by an IPulserReceiverIdentity instance which contains information about the Instrument that owns it and the 0-based numeric index within the Instrument. This instance can be used to select the current Pulser-Receiver in the JSRDotNETManager or retrieve the IPulserReceiver instance from the JSRDotNETManager for directly controlling the Pulser-Receiver device.

IInstrumentIdentity – This interface identifies a specific Instrument attached to the Client PC. This object contains the Model Number, Serial Number and Port for the Instrument. This can be used with the JSRDotNETManager to retrieve the list of Pulser-Receivers that are owned by this instrument. Most Instruments contain either one or two Pulser-Receivers.

IPulserReceiverIdentity – This interface identifies a specific Pulser-Receiver owned by an Instrument attached to the Client PC. This object contains the IInstrumentIdentity of the parent Instrument and the Pulser-Receiver index within that instrument.

IPulserReceiver – This interface defines a set of Properties and Methods for controlling a Pulser-Receiver device and retrieving its current configuration. An IPulserReceiverIdentity instance is required for retrieving a specific IPulserReceiver instance. The JSRDotNETManager also implements this interface and can control a Pulser-Receiver in the same way as using an IPulserReceiver instance directly.

JSRDotNETManager.GetInstruments() – This method returns a list of the Instruments attached to the Client PC that have been discovered. Keep in mind that the JSRDotNETManager will only discover devices that are supported by the Instrument Plugins in the Managed Plugins list.

JSRDotNETManager.GetPulserReceivers(IInstrumentIdentity) – This method returns a list of IPulserReceiverIdentity instances for the Pulser-Receivers that belong to the Instrument identified by the provided IInstrumentIdentity instance.

JSRDotNETManager.GetPulserReceivers() – This method returns a list of IPulserReceiverIdentity instances for all of the Pulser-Receivers discovered by the JSRDotNETManager.

JSRDotNETManager.GetPulserReceiver(IPulserReceiverIdentity) – Returns a single IPulserReceiver instance identified by the provided IPulserReceiverIdentity instance. The IPulserReceiver instance can be used to retrieve and set the current Pulser-Receiver configuration.

JSRDotNETManager.SetCurrentPulserReceiver(IPulserReceiverIdentity) – Sets the Pulser-Receiver device that will be represented by the JSRDotNETManager. After setting the current Pulser-Receiver, the JSRDotNETManager can be used to access all of the Methods and Properties that are provided by the IPulserReceiver interface.

## Working with Pulser-Receiver Instances

There are two methods for querying and controlling a Pulser-Receiver device from your application:

1. *JSRDotNETManager method* – Use the SetCurrentPulserReceiver() method to select a Pulser-Receiver device to control and use the JSRDotNETManager to get and set the different properties of the device. Since the JSRDotNETManager implements the IPulserReceiver interface, it provides the same methods and properties as a Pulser-Receiver instance.
   1. The JSRDotNetManager will perform checks to ensure the pulser is still present, and then dispatches the request to a background thread to process. The calling thread is blocked for the duration of the operation.
   2. Only one instrument can be selected at a time.
   3. This method is more secure for the calling thread and is multi-thread safe, but slower.
2. *Pulser-Receiver Instances method* – Retrieve a Pulser-Receiver instance from the JSRDotNETManager by calling the GetPulserReceiver() method. The instance can then be used to get and set the different properties of the device.
   1. The IPulserReceiver interface performs no checks for the presence of the instrument and will interact with the instrument directly from the calling thread.
   2. Each IPulserReceiver instance should interact with only one calling thread at a time. It performs limited thread safety management.
   3. Separate IPulserReceiver instances will operate in parallel without an issue, so an application can manage multiple instruments at the same time.
   4. This method is the fastest interface, but additional design considerations may be required in the application to manage instrument conditions and thread safety.

**NOTE:** Your application should not attempt to use both the *JSRDotNETManager* and the *Pulser-Receiver Instances* methods at the same time or your application may experience undesired behaviors.

**NOTE:** Calling either method directly from a UI thread may cause unwanted delays. Both methods will block the calling thread in order to make the operation appear synchronous to the caller even though the operation is actually asynchronous within the system. Instrument response times will vary by model type, and thread queuing in the *JSRDotNETManager* method produces additional overhead and thread scheduling delays from the operating system. The creation of a background thread to manage the device interaction may be useful to keep a UI thread fully responsive.

The following examples illustrate the differences between the two methods. This example retrieves the current Gain, subtracts 1.0dB and then sets the new Gain value. These examples assume the desired Pulser-Receiver has already been selected and an IPulserReceiverIdentity object is available for this device.

**JSRDotNETManager method**

//IPulserReceiverIdentity prId – Previously set for this example

// Set this Pulser-Receiver as the current device - this function will throw the

// CURRENT\_PULSER\_RCVR\_CHANGED event.

jsrManager**.**SetCurrentPulserReceiver**(**prId**);**

// Get the current gain from the Pulser-Receiver selected in the Manager

double gain1 **=** jsrManager**.**Gain**;**

Console**.**WriteLine**(**"Current Gain: {0}"**,** jsrManager**.**Gain**);**

gain1 **+=** 1.0d**;**

// Use the Manager to set the current gain on the selected Pulser-Receiver

jsrManager**.**Gain **=** gain1**;**

Console**.**WriteLine**(**"New Gain: {0}"**,** jsrManager**.**Gain**);**

**Pulser-Receiver Instances method**

//IPulserReceiverIdentity prId – Previously set for this example

// Get the Pulser-Receiver instance from the JSRDotNETManager

IPulserReceiver prDevice **=** jsrManager**.**GetPulserReceiver**(**prId**);**

// Get the current gain from the Pulser-Receiver instance

double gain2 **=** prDevice**.**Gain**;**

Console**.**WriteLine**(**"Current Gain: {0}"**,** prDevice**.**Gain**);**

// Add 1.0dB

gain2 **+=** 1.0d**;**

// Set the new gain on the Pulser-Receiver instance

prDevice**.**Gain **=** gain2**;**

Console**.**WriteLine**(**"New Gain: {0}"**,** prDevice**.**Gain**);**

## Other Methods for Querying and Controlling a Pulser-Receiver

There are three main techniques for querying and controlling a Pulser-Receiver, regardless of whether the JSRDotNETManager or IPulserReceiver instance methods are used. The following sections provide information for each of these techniques.

1. **Get/Set Properties**

The IPulserReceiver interface defines a common set of Properties for a Pulser-Receiver. Since the list of Properties is the same for all Pulser-Receivers types, but are not supported by every Pulser-Receiver models, additional Properties have been provided to return whether or not the Property is supported as well as the supported value ranges or lists. The following example illustrates how to set the Gain using the Properties:

**try**

**{**

// Check to see if a list of gain values is supported

**if** **(**m\_jsrManager**.**GainIndexSupported**)**

**{**

double**[]** gainValues **=** m\_jsrManager**.**GainValues**;**

int i**;**

**for** **(**i **=** 1**;** i **<** gainValues**.**Length**;** i**++)**

**{**

// Find the first index where newGain is less than the gain value

**if** **(**newGain **<** gainValues**[**i**])**

**break;**

**}**

// i will be in the range of 1 to gainValues.Length, inclusive

// use the value that is 1 below the last value for i

m\_jsrManager**.**GainIndex **=** **(**i **-** 1**);**

bSuccess **=** **true;**

**}**

// Otherwise, make sure the gain is within the Min and Max values

**else** **if** **((**newGain **>=** m\_jsrManager**.**GainMin**)** **&&** **(**newGain **<=** m\_jsrManager**.**GainMax**))**

**{**

// Now set the gain directly

m\_jsrManager**.**Gain **=** newGain**;**

bSuccess **=** **true;**

**}**

**}**

**catch** **(**ExceptionJSRDotNET**)**

**{**

// Exception occurred – TODO: Print something helpful...

bSuccess **=** **false;**

**}**

1. **Get/SetPulserPropertyValue Methods**

The GetPulserPropertyValue and SetPulserPropertyValue methods are similar to the functions provided in the original JSR Common API and provide an indirect method for retrieving and setting the Pulser-Receiver configuration. These methods return a generic object type, which must be cast to the appropriate type (string, double, int, string[], double[], int[], etc.) and C# provides methods for determining the type of the value returned. A try-catch block is recommended for these methods since an ExceptionJSRDotNET will be thrown if the property name is not supported or found or if the Pulser-Receiver does not support setting the property value. These functions also support retrieving and setting the values for custom properties that are supported by the Pulser-Receiver (e.g. “IsPulserPresent” for the DPR500 Pulser-Receivers). The following example takes the same Gain operation as before and uses the GetPulserPropertyValue and SetPulserPropertyValue methods to retrieve the valid value ranges and set the new Gain value.

**try**

**{**

// Check to see if a list of gain values is supported

**if** **((**bool**)**m\_jsrManager**.**GetPulserPropertyValue**(**PRPropertyNames**.**GainIndexSupported**))**

**{**

double**[]** gainValues **=**

m\_jsrManager**.**GetPulserPropertyValue**(**PRPropertyNames**.**GainValues**)**

**as** double**[];**

**if** **((**gainValues **!=** **null)** **&&** **(**gainValues**.**Length **>** 0**))**

**{**

int i**;**

**for** **(**i **=** 1**;** i **<** gainValues**.**Length**;** i**++)**

**{**

// Find the first index where newGain is less than the gain value

**if** **(**newGain **<** gainValues**[**i**])**

**break;**

**}**

// i will be in the range of 1 to gainValues.Length, inclusive

// use the value that is 1 below the last value for i

m\_jsrManager**.**SetPulserPropertyValue**(**PRPropertyNames**.**GainIndex**,** **(**i **-** 1**));**

bSuccess **=** **true;**

**}**

**}**

// Otherwise, make sure the gain is within the Min and Max values and set directly

**if** **(!**bSuccess**)**

**{**

double gainMin **=**

**(**double**)**m\_jsrManager**.**GetPulserPropertyValue**(**PRPropertyNames**.**GainMin**);**

double gainMax **=**

**(**double**)**m\_jsrManager**.**GetPulserPropertyValue**(**PRPropertyNames**.**GainMax**);**

**if** **((**newGain **>=** gainMin**)** **&&** **(**newGain **<=** gainMax**))**

**{**

// Now set the gain directly

m\_jsrManager**.**SetPulserPropertyValue**(**PRPropertyNames**.**Gain**,** newGain**);**

bSuccess **=** **true;**

**}**

**}**

**}**

**catch** **(**ExceptionJSRDotNET**)**

**{**

// Exception occurred – TODO: Print something helpful...

bSuccess **=** **false;**

**}**

1. **PulserSettingInfo Instances**

The PulserSettingInfo instances provide an efficient way for retrieving information about the different settings supported by a Pulser-Receiver. A single PulserSettingInfo instance contains access for querying whether the setting is Direct (set the value directly) or Indexed (set the value as an index into an array of values) and retrieving the value. There are also helper functions that retrieve the value as a string and the Value property which allows your program to set a value directly, regardless of whether it is Direct or Indexed. Each IPulserReceiver instance will return a list of supported settings as PulserSettingInfo instances. This includes any standard settings or custom settings that are not included in the IPulserReceiver interface. The following example demonstrates how easy it is to set the gain value using the PulserSettingInfo instance. Since each PulserSettingInfo info instances is connected directly to the currently selected Pulser-Receiver in the JSRDotNETManager, the PulserSettingInfo dictionary will need to be repopulated every time the currently selected Pulser-Receiver is changed.

// Store the PulserSettingInfo objects in a Dictionary for easy access

System**.**Collections**.**Generic**.**Dictionary**<**string**,** PulserSettingInfo**>** dictSettings **=**

**new** System**.**Collections**.**Generic**.**Dictionary**<**string**,** PulserSettingInfo**>();**

...

// When a Pulser-Receiver is selected, populate the PulserSettingInfo dictionary

**try**

**{**

dictSettings**.**Clear**();**

**foreach** **(**PulserSettingInfo setting **in** m\_jsrManager**.**PulserSettings**)**

**{**

dictSettings**.**Add**(**setting**.**SettingName**,** setting**);**

**}**

**}**

**catch(**Exception ex**)**

**{**

Console**.**WriteLine**(**"Error retrieving Pulser Settings: " **+** ex**.**Message**);**

**}**

...

**try**

**{**

**if** **(**dictSettings**.**ContainsKey**(**SettingNames**.**Gain**))**

**{**

// Get the PulserSettingInfo instance for the Gain...

PulserSettingInfo gainSetting **=** dictSettings**[**SettingNames**.**Gain**];**

**if** **(**gainSetting **!=** **null)**

**{**

// Set the Gain to the new value...

gainSetting**.**Value **=** newGain**;**

bSuccess **=** **true;**

**}**

**}**

**}**

**catch** **(**ExceptionJSRDotNET**)**

**{**

// Exception occurred – TODO: Print something helpful...

bSuccess **=** **false;**

**}**

# APPENDIX A – Sample application source code

The source code for the application developed in this document is contained in the JSRDotNET\_Sample project in the JSRDotNET\_Examples solution.

////////////////////////////////////////////////////////////////////////////////////////////////////

// JSRDotNET\_Sample

//

// summary: The source of the sample application that is described in the

// JSRDotNETSDK\_QuickStart documentation.

//

// Copyright (C) 2016, BYK-Garnder USA

//

// See the file "LICENSE\_EXAMPLES.TXT" for the full license governing this code.

////////////////////////////////////////////////////////////////////////////////////////////////////

**using** System**;**

**using** System**.**Reflection**;**

**using** System**.**Threading**;**

**using** JSRDotNETSDK**;**

**namespace** JSRDotNET\_Sample

**{**

class SampleProgram

**{**

#region Local Variables

/// <summary>true if error occurred.</summary>

**private** bool m\_bErrorOccurred **=** **false;**

/// <summary>The wait event used for device discovery</summary>

**private** AutoResetEvent m\_WaitEvent **=** **new** AutoResetEvent**(false);**

#endregion

#region Main() Entry Function

///////////////////////////////////////////////////////////////////////////////////////////////

/// <summary>Main entry-point for this application.</summary>

///

/// <remarks>mwb, 5/6/2016.</remarks>

///

/// <param name="args">Array of command-line argument strings.</param>

///////////////////////////////////////////////////////////////////////////////////////////////

static void Main**(**string**[]** args**)**

**{**

SampleProgram myProgram **=** **new** SampleProgram**();**

// Start the program execution

myProgram**.**Run**();**

**}**

#endregion

#region Main Application Method

///////////////////////////////////////////////////////////////////////////////////////////////

/// <summary>Runs this object.</summary>

///

/// <remarks>mwb, 5/6/2016.</remarks>

///////////////////////////////////////////////////////////////////////////////////////////////

**internal** void Run**()**

**{**

// We need the path to the plugins directory that contains the JSRDotNET

// plugins. In this case, the plugins are in a subdirectory where the

// executable is located. Use the Assembly functions to get the application

// directory.

string pluginPath **=** System**.**IO**.**Path**.**GetDirectoryName**(**

Assembly**.**GetExecutingAssembly**().**Location**)** **+** @"\plugins"**;**

// Create a new JSRDotNETManager using the default constructor

JSRDotNETManager jsrManager **=** **new** JSRDotNETManager**(**pluginPath**);**

// OR

// Create a new JSRDotNETManager using the default constructor

// JSRDotNETManager jsrManager\_alt = new JSRDotNETManager();

// jsrManager\_alt.PluginPath = pluginPath;

**try**

**{**

// Load all of the Plugin dlls - this may throw an exception

// if there is an error with the path.

jsrManager**.**LoadPlugins**();**

**}**

**catch** **(**Exception ex**)**

**{**

Console**.**WriteLine**(**"Error loading plugins: " **+** ex**.**Message**);**

**}**

// At this point the jsrManager has a reference to the plugins that were

// discovered in the given directory.

// Get the list of plugins found (these are just the names)

**foreach** **(**string name **in** jsrManager**.**GetPluginNames**())**

**{**

Console**.**WriteLine**(**"Found plugin: '" **+** name **+** "'"**);**

// Get the metadata (info) about the plugin

IJSRDotNETLibMetadata pluginInfo **=** jsrManager**.**GetPluginLibraryMetadata**(**name**);**

// Print some info about the plugin

Console**.**WriteLine**(**" - " **+** pluginInfo**.**FriendlyName **+** " - " **+**

string**.**Join**(**", "**,** pluginInfo**.**SupportedModels**)** **+**

" (" **+** string**.**Join**(**", "**,** pluginInfo**.**ConnectionType**)** **+** ")"**);**

jsrManager**.**AddManagedPlugin**(**name**);**

**}**

// Add our NotifyEventHander method to the jsrManager

jsrManager**.**NotifyEventHandler **+=** **this.**OnJsrManagerNotify**;**

// Start the device discovery

jsrManager**.**SetDiscoveryEnable**(true);**

Console**.**WriteLine**();**

Console**.**WriteLine**(**"Waiting for devices... Press any key to stop"**);**

Console**.**WriteLine**();**

int sleepWait **=** 20000**;**

**while** **(**sleepWait **>** 0**)**

**{**

sleepWait **-=** 100**;**

**if** **((**m\_WaitEvent**.**WaitOne**(**100**))** **||** **(**Console**.**KeyAvailable**))**

sleepWait **=** 0**;**

**}**

**while** **(**Console**.**KeyAvailable**)**

Console**.**ReadKey**(true);**

**if** **(!**m\_bErrorOccurred**)**

**{**

IInstrumentIdentity**[]** instrList **=** jsrManager**.**GetInstruments**();**

**if** **((**instrList **!=** **null)&&(**0 **<** instrList**.**Length**))**

**{**

**foreach** **(**IInstrumentIdentity instrId **in** instrList**)**

**{**

Console**.**WriteLine**(**

"============================================================="**);**

Console**.**WriteLine**(**" INSTRUMENT: {0}[{1}] - Port: {2} ({3})"**,**

instrId**.**ModelName**,** instrId**.**SerialNum**,** instrId**.**Port**,** instrId**.**PluginName**);**

Console**.**WriteLine**(**

"============================================================="**);**

Console**.**WriteLine**();**

// Get the list of the discovered pulser/receivers, select each one a print

// out some parameters for each.

**foreach** **(**IPulserReceiverIdentity prID **in** jsrManager**.**GetPulserReceivers**(**instrId**))**

**{**

Console**.**WriteLine**(**

"-------------------------------------------------------------"**);**

Console**.**WriteLine**(**" P/R: {0}[{1}] - IDX {2}"**,**

prID**.**InstrumentId**.**ModelName**,** prID**.**InstrumentId**.**SerialNum**,**

prID**.**PulserReceiverIndex**);**

Console**.**WriteLine**(**

"-------------------------------------------------------------"**);**

**try**

**{**

// Set this pulser/receiver as the current device - this function will

// throw the CURRENT\_PULSER\_RCVR\_CHANGED event.

jsrManager**.**SetCurrentPulserReceiver**(**prID**);**

// We can now call all the functions available in the IPulserReceiver

// interface on the selected pulser/receiver by using the

// JSRDotNETManager

Console**.**WriteLine**(**" BANDWIDTH: {0:F}"**,** jsrManager**.**MaxFrequency**);**

Console**.**WriteLine**(**" SIGNAL\_SELECT: {0}"**,**

jsrManager**.**ReceiverMode**.**ToString**());**

Console**.**WriteLine**(**" GAIN: {0:F2}"**,** jsrManager**.**Gain**);**

Console**.**WriteLine**(**" LOW\_PASS: {0:F0}"**,**

jsrManager**.**LowPassFilterValues**[**jsrManager**.**LowPassFilterIndex**]);**

Console**.**WriteLine**(**" HIGH\_PASS: {0:F0}"**,**

jsrManager**.**HighPassFilterValues**[**jsrManager**.**HighPassFilterIndex**]);**

Console**.**WriteLine**(**" PRF: {0:F0}"**,**

jsrManager**.**PulseRepetitionFrequency**);**

Console**.**WriteLine**(**" ENERGY\_PER\_PULSE: {0:F2}"**,**

jsrManager**.**EnergyPerPulse**);**

Console**.**WriteLine**(**" VOLTAGE: {0:F0}"**,** jsrManager**.**HVSupply**);**

**if** **(**jsrManager**.**HVSupplyEnableSupported**)**

Console**.**WriteLine**(**" HV\_SUPPLY: {0:F0}"**,**

**(**jsrManager**.**HVSupplyEnable **?** "ENABLED" **:** "DISABLED"**));**

**if** **(**jsrManager**.**PulseEnergyIndexSupported**)**

Console**.**WriteLine**(**" ENERGY\_CONTROL: {0}"**,**

jsrManager**.**PulseEnergyValueNames**[**jsrManager**.**PulseEnergyIndex**]);**

**if** **(**jsrManager**.**DampingIndexSupported**)**

Console**.**WriteLine**(**" DAMPING: {0}"**,**

jsrManager**.**DampingValues**[**jsrManager**.**DampingIndex**]);**

**if** **(**jsrManager**.**PulserImpedanceSupported**)**

Console**.**WriteLine**(**" PULSER\_IMPEDANCE: {0}"**,**

jsrManager**.**PulserImpedance**.**ToString**());**

Console**.**WriteLine**(**" TRIGGER\_SOURCE: {0}"**,**

jsrManager**.**TriggerSource**.**ToString**());**

**if** **(**jsrManager**.**TriggerEdgePolaritySupported**)**

Console**.**WriteLine**(**" TRIGGER\_POLARITY: {0}"**,**

jsrManager**.**TriggerEdgePolarity**.**ToString**());**

**if** **(**jsrManager**.**TriggerImpedanceSupported**)**

Console**.**WriteLine**(**" TRIGGER\_IMPEDANCE: {0}"**,**

jsrManager**.**TriggerImpedance**.**ToString**());**

Console**.**WriteLine**(**" IS\_TRIGGERED: {0}"**,**

**(**jsrManager**.**TriggerEnable **?** "YES" **:** "NO"**));**

Console**.**WriteLine**(**" IS\_PULSING: {0}"**,**

jsrManager**.**PulserIsPulsing**.**ToString**());**

**if** **(**jsrManager**.**PulserTriggerCountSupported**)**

Console**.**WriteLine**(**" TRIGGER\_COUNT: {0:D}"**,**

jsrManager**.**PulserTriggerCount**);**

Console**.**WriteLine**(**" POWER\_LIMIT: {0}"**,**

jsrManager**.**PulserPowerLimitStatus**.**ToString**());**

**if** **(**jsrManager**.**LEDBlinkModeIndexSupported**)**

Console**.**WriteLine**(**" BLINK\_MODE: {0}"**,**

jsrManager**.**LEDBlinkModeValues**[**jsrManager**.**LEDBlinkModeIndex**]);**

**}**

**catch** **(**Exception ex**)**

**{**

Console**.**WriteLine**(**"Exception occurred during processing: " **+**

ex**.**Message**);**

**}**

Console**.**WriteLine**();**

**}** // for(pulserReceivers... )

**}**// for(instruments... )

**}** // If count > 0

**else**

**{**

Console**.**WriteLine**();**

Console**.**WriteLine**(**" No Pulser Receivers found."**);**

Console**.**WriteLine**();**

**}**

**}** // if !m\_bErrorOccurred

// Make sure device discovery is turned off

jsrManager**.**SetDiscoveryEnable**(false);**

// Unregister our NotifyEventHander

jsrManager**.**NotifyEventHandler **-=** **this.**OnJsrManagerNotify**;**

Console**.**WriteLine**(**"Press any key to exit..."**);**

Console**.**ReadKey**(true);**

// Make sure any local references to pulser/receivers have been freed...

// Application is finished with the JSRDotNETManager; call the Shutdown() function

jsrManager**.**Shutdown**();**

**}**

#endregion

#region JSRDotNETManager Event Handlers

///////////////////////////////////////////////////////////////////////////////////////////////

/// <summary>Event handler for notify events from the JSRDotNETManager.</summary>

///

/// <remarks>mwb, 4/1/2016.</remarks>

///

/// <param name="sender">Source of the event.</param>

/// <param name="e">The EventArgsManagerNotify to process.</param>

///////////////////////////////////////////////////////////////////////////////////////////////

**internal** void OnJsrManagerNotify**(object** sender**,** EventArgsManagerNotify e**)**

**{**

// We don't care about this notification

Console**.**WriteLine**();**

Console**.**WriteLine**(**"EVENT{0}: {1}"**,**

**(**string**.**IsNullOrWhiteSpace**(**e**.**PluginName**)** **?** "" **:** "[" **+** e**.**PluginName **+** "]"**),**

e**.**NotifyType**);**

**if** **(**e**.**PulserReceiverId **!=** **null)**

**{**

Console**.**WriteLine**(**"DEVICE: {0}"**,** e**.**PulserReceiverId**.**ToString**());**

**}**

**if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**PULSER\_RCVR\_DETACH**)**

**{**

**if** **(**e**.**WasSelected**)**

Console**.**WriteLine**(**"DETACHED: Device was the selected device"**);**

**}**

**else** **if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**DISCOVERY\_COMPLETE**)**

**{**

m\_WaitEvent**.**Set**();**

**}**

**else** **if** **(**e**.**NotifyType **==** NOTIFY\_TYPE**.**ERROR**)**

**{**

**if** **(**0 **==** **(**e**.**DiscoverState **&** DiscoveryStateFlags**.**PENDING**))**

**{**

Console**.**WriteLine**(**" ERROR MSG: " **+** e**.**ErrorMsg**);**

Console**.**WriteLine**(**"ERROR TEXT: " **+** e**.**ErrorText**);**

m\_bErrorOccurred **=** **true;**

m\_WaitEvent**.**Set**();**

**}**

**else**

**{**

Console**.**WriteLine**();**

Console**.**WriteLine**(**e**.**ErrorMsg**);**

Console**.**WriteLine**();**

**}**

**}**

**}**

#endregion

**}**

**}**

# APPENDIX B – Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Rev | Release Date | Description of Change | Modified By |
| 0.1.0 | 4/6/2016 | Initial Release – Alpha Software Release | Mike Wilkerson-Barker  Kevin Stokes |
| 0.8.0 | 5/18/2016 | SDK Updates for Beta Software Release (0.8.0), changed title and name of file. | Mike Wilkerson-Barker |
| 0.8.1 | 6/28/2016 | Updates for Release Candidate 1 Release | Mike Wilkerson-Barker |
| 0.8.2 | 7/11/2016 | Updates for Release Candidate 2 Release | Mike Wilkerson-Barker |
| 1.0.0 | 4/11/2018 | Updates for 1.0.0 Release | M. Hubbard |
| 1.0.1 | 5/24/2019 | Updates for 1.0.1 Patch | M. Hubbard |
| 1.2.8 | 4/3/2023 | Updated version number | K. Stokes |
| 1.2.9 | 8/10/2023 | Updated version number | K.Stokes |
| 1.2.12 | 2/06/2024 | Updated version number and dates | K. Stokes |
| 1.2.13 | 5/29/2024 | Updated version number and dates | K.Stokes |