(.NET SDK) Getting Started

External Data

Monnit Corporation

Version 3.2.0

Contents

Introduction

Monnit Mine SDK was created to allow you to easily incorporate Monnit wireless sensors and gateways into your existing applications. The Monnit Mine SDK can be located at mine.imonnit.com. The sample application is a Windows Form program that is meant to help the user understand the use of the Mine SDK. By inspecting the code executed from the inputs of the form, you should be getting an understanding of what code, from the Mine Library, was necessary in order to execute the function of inspected input.

This example of the Mine SDK is an application that allows communication to an external data source, e.g., a SQL Database. This sample app reads and writes from a JSON file, which can be swapped out and modified for a database interface that you have built.

If you're looking for a simpler application, that is a stand-alone application in which you manipulate data within the application itself. Look to utilize our other Sample Application.

Requirements

At the current release, the SDK is provided as a .NET library. Your platform will need to be able to host this library. Known good configurations include running the application in a Microsoft Windows environment with the .NET Framework installed. Other configurations may work such as Linux OS running Mono, but these have not been tested by Monnit at this stage.

To use a Monnit Gateway with Mine you will need to purchase an unlock code to be able to point the gateway to the Mine Server you create. You can purchase the unlock code at www.monnit.com.

Sample projects have been built using Visual Studio 2019 and can be compiled and run using the Free Express version available from Microsoft here: https://visualstudio.microsoft.com/vs/express/

Overview

After purchasing your Monnit sensors you are provided a free 45 day trial of our online platform, iMonnit (www.imonnit.com). All sensors and gateways are preconfigured to work with this platform making it easy to get things up and running. We recommend you become familiar with some of the basic use of the hardware using this portal first. After becoming familiar with how the sensors work with the gateways you'll find it easier to implement your solution.

Many questions can be answered by searching the online knowledge base or visiting our FAQ page: http://www.monnit.com/support/frequently-asked-questions

Monnit support staff is well versed in use of the sensors with our provided monitoring platforms. For support of sensors and gateways they will ask you to troubleshoot them using the iMonnit platform. A select few members of the support staff are able to respond to questions directly involving the SDK. Your best option for SDK support is to email support@monnit.com and your question will be directed to a member of the support staff that will be able to respond. We do strive to reply to all questions in a reasonable timeframe however, the response time for SDK questions may take longer than sensor or gateway related questions.

Basic Usage

The following will outline the basic use of the SDK to help you get your code running with minimal effort. It will loosely follow the code in the sample application without the specific UI constraints presented in the example.

Start by adding references to bring the Monnit Mine SDK library into your project. Then add any needed Using statements to the top of your code file. Some of the namespaces we've included may not be needed for your implementation.

Code Example:

```
using System;
using System.Collections.Generic;
using System.NET;
using System.NET.NETworkInformation;
using System.NET.Sockets;
using Monnit;
using Monnit.Mine;
```

Initializing the Server Class

After including the library, create an instance of the server class. Each instance can run independently and can be configured to its own IP Address, protocol, and port. The simplest implementation is to use a single instance to bind to all interfaces.

To instantiate a MineServer object, first choose the protocol to use, TCP and/or UDP. The types of gateways you have will determine which protocol is needed. TCP is used for Ethernet and USB Gateways and UDP for Cellular gateways. Second, select the computers IP Address you would like to use. (System.NET.IPAddress.Any will bind to all available IP Addresses on the computer.) Thirdly, select the port, documentation will use the default port of 3000. Lastly, pass in an instance of an implementation of the interface iDataAccess. The sample app comes with an implementation called ExternalDataAccess.cs; it interacts with a JSON file in order to store device information.

You'll need to keep a reference to the server and data access instance as you will be interacting with it later in your application.

Code Example:

```
ExternalDataAccess DataAccess = new ExternalDataAccess();
MineServer ServerInstance = new MineServer(eMineListenerProtocol.TCPAndUDP, IPAddress.Any, 3000,
DataAccess);
ServerInstance.StartServer();
```

After the server instance is created you can start the socket listeners using the StartServer method.

Subscribing to Events

The MineServer object will fire events when certain things happen. For instance, when a gateway delivers a message, the "GatewayMessage" event is fired. If the message contains sensor data then additionally the "SensorMessage" event is fired. There is no need to subscribe to all events, just the ones you want to handle.

Code Example:

```
ServerInstance.LogException += Handle_LogException;
ServerInstance.PersistGateway += Handle_PersistGateway;
ServerInstance.PersistSensor += Handle_PersistSensor;
ServerInstance.SensorMessage += Handle_SensorMessage;
ServerInstance.GatewayMessage += Handle_GatewayMessage;

void Handle_GatewayMessage(object sender, GatewayMessageEventArgs e) {}

void Handle_SensorMessage(object sender, SensorMessageEventArgs e) {}

void Handle_PersistSensor(object sender, HandlePersistSensorEventArgs e) {}

void Handle_PersistGateway(object sender, HandlePersistGatewayEventArgs e) {}

void Handle_LogException(object sender, HandleLogExceptionEventArgs e) {}
```

Managing Gateways

Now that your server is operating and is ready to deliver your event data, you need to add your gateway(s) information to your external data source so that when you call FindGateway the server knows what type and version of gateway is talking to it. This is so it can properly interpret the data which the gateway will send.

You'll need to instantiate an instance of the Gateway class from the data you retrieve from your external data source. This allows the values to be up to date each time the server calls FindGateway.

Code Example:

```
Gateway MineGateway = new Gateway(12345, eGatewayType.USB, "3.3.2.1", "2.3.0.0", "127.0.0.1", 3000);
```

Now you can start interacting with your gateway and receiving messages from it.

Another overload to be aware of is the one with these parameters

- EncryptionKey
    - Encryption key generated during handshake, if server is restarted or multiple server instances are configured this key needs to match across all instances or the gateway will have to re initialize its security handshake
- EncryptionIV
    - Initialization vector generated during handshake, if server is restarted or multiple server instances are configured this key needs to match across all instances or the gateway will have to re initialize its security handshake
- Keep in mind when you use these 2 parameters, you'll have to populate the 'PersistGateway' event for your MineServer instance to persist any gateway changes/updates to your external storage.

Managing Sensors

The server will also need to know some information about the sensors in order to properly interact with them. This is done in the same manner as the gateway by creating a data record of a sensor object from your external data store. Like the gateway there is a basic constructor with just the essential information, and a secondary that also configures one or more of the optional parameters. The sensor object is assigned to one of the gateways by having the sensor ID on the gateways sensor list in your external data store. This lets the server know which gateway(s) the sensor is configured to communicate through. If you would like the sensor to be assigned to multiple gateways you can simply assign the same sensor ID with additional Gateway's sensor list. This feature allows you to deploy multiple gateways over a large area and the sensors will be able to communicate with whichever gateway is in range.

Code Example:

```
Sensor MineSensor = new Sensor(54321, eSensorApplication.Temperature, "2.3.0.0", "GEN1");
```

If you are planning on using the SensorPrint feature, please refer to the SensorPrint section on page 9 for additional steps in registering your sensor.

Other Usage Notes

The basic usage will get you up and running with the Monnit Mine SDK. However, there are more things you may want to be aware of.

Device Behavior

Over the course of use some information in the device can be updated. For instance, when an Ethernet gateway sends in its startup message it delivers its MAC address and updates the gateway object. One of the events you find that the server will fire is “PersistGateway”. After information in the gateway object is updated this event fires and allows you to store this new information into your external data store for use, or for you to use with subsequent instantiations. There is a similar event “PersistSensor” that is used in the same manner. These are optional depending on your level of integration with the system.

Device Lookups

You can retrieve an instance of your gateway or sensor object by using the servers “Find” methods. If you are observing the “PersistGatway” event you will see that it sends the Gateway object.

Code Example:

```
void Handle_PersistGateway(object sender, HandlePersistGatewayEventArgs e)
{
     PrintOutput(string.Format("GatewayID {0} was updated in memory.", e.Gateway.GatewayID));
     DataAccess.UpdateGatewayEncryption(e.Gateway);
}
```

Stopping the Server

The server also has a “StopServer” method which will close and dispose of the sockets. (*Note: Does not affect the gateways flash memory; see Reforming Network on a gateway.*)

Reforming a Network

Reforming a gateways network does a couple of things.

First, it tells the gateway to scan for an open channel in the area. This can assist in obtaining optimal range, but by selecting a new channel any sensors that have been linked to the gateway will have to fail out and rescan to find the gateway on its newly selected channel. In general, you don’t want to reform the gateway very often.

Second it clears the sensors from the gateways flash memory and downloads a new sensor list from the server, which is actually grabbing the sensor list from your external data store.

This is important because if you had a sensor assigned to Gateway_A but want to change the sensor to communicate through Gateway_B in the same area, even after removing the sensor from Gateway_A and assigning it to Gateway_B, the sensor will still be able to communicate through either gateway. This is because it still exists in the flash memory of both gateways. To force it to no longer communicate with Gateway_A you will need to reform Gateway_A so the sensor is no longer allowed to join forcing it to now communicate only through Gateway_B. Optionally you can just allow it to talk to either gateway by assigning it to both Gateway_A and Gateway_B.

Update Sensor

To update a sensor you will need to know the sensors application type by calling ApplicationBase.GetType(int applicationID) and passing the sensors `MonnitApplication and casting it as an int` this will bring back the correct class, so that you can call the correct sensor edit function and update page you will want to create. Looking at TemperatureBase's SensorEdit function we can see that it contains the following parameters:

1. Sensor sens
2. double? Heartbeat
3. double? AwareStateHeartBeat
4. int? assessmentsPerHeartBeat
5. double? minimumThreshold
6. double? maximumThreshold
7. double? Hysteresis
8. int? failedTransmissionBeforeLinkMode

Code Example:

```
void button1_click(object sender, EventArgs e)
{
     double? hb = double.Parse(hbtxtbx.Text);
     double? ahb = double.Parse(ashtxtbx.Text);
     int? aphb = int.Parse(aphtxtbx.Text);
     double? min = double.Parse(minthreshtxtbx.Text);
     double? max = double.Parse(maxthreshtxtbx.Text);
     double? hyst = double.Parse(hysttxtbx.Text);
     int? failedtrans = double.Parse(ftblmtxtbx.Text);

     try
     {
          Monnit.TemperatureBase.SensorEdit(sensor, hb, ahb, aphb, min, max, hyst, failedtrans);
     }
     catch
     {
          MessageBox.Show("Failed to update");
     }
}
```

We allow for nullable fields if you do not wish to update a specific parameter.

In order to update other sensor types, you'll have to find the 'eSensorApplication' that correlates with the sensor you're using. So, say you want to use a Button sensor. You'll want to create a 'UpdateButtonSensor.cs' file, I would recommend copying the 'UpdateTemperatureSensor.cs' file. Then modifying it to fit the parameters needs of the ButtonLedBase.SensorEdit() method.

Code Example:

```
void button1_click(object sender, EventArgs e)
{
     double? heartbeat = double.Parse(hbtxtbx.Text);
     double? awareHeartBeat = double.Parse(ashtxtbx.Text);
     int? enterAwareState = int.Parse(easwli.Text);
     int? rearmTime = int.Parse(rearmTime.Text);
     int? failedtrans = double.Parse(ftblmtxtbx.Text);

     try
     {
          ButtonLedBase.SensorEdit(sensor, heartbeat, awareHeartBeat, enterAwareState, rearmTime,
                                   failedtrans);
     }
     catch
     {
          MessageBox.Show("Failed to update");
     }
}
```

Sensor Print

To use the Sensor Print feature with Mine you will need to purchase Sensor Print credit(s) from www.monnit.com to be able to utilize the feature on your sensor(s). In short, what this feature does is it adds a layer of communication security between your sensor and server.

If you have purchased the feature for a sensor, in order to properly use the Sensor Print feature make sure to follow the code example below when registering your sensor.

Code Example:

```
Monnit.Mine.Sensor MineSensor = new Monnit.Mine.Sensor(SensorID, SensorApplication, "2.3.0.0",
generation);
// To apply SensorPrint, you need to manually assign the property.
// Hex string must be at a length of 64, and Hexadecimal values only (0-9, A-F)
string sensorPrintHexString = "1234123412341234123412341234123412341234123412341234123412341234";
byte[] sensorPrintByteArray = ExtentionMethods.StringToByteArray(sensorPrintHexString);
MineSensor.SensorPrint = sensorPrintByteArray;
```

Now in your SensorMessage handler the SensorMessage object will have a flag '`IsAuthenticated`', meaning that the message from the sensor is digitally signed and validated at the server.