(Java SDK) Getting Started

Monnit Corporation

Version 4.0.0.1

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 downloaded from mine.imonnit.com. The sample application is a Java Swing GUI 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.

If you're wanting to integrate the Mine SDK into an application that allows communication to an external data source, e.g., a SQL Database. You should look at the External Data sample application that we provide. As that example reads and writes from a JSON file, which can be swapped out and modified for a database interface that you have built.

Requirements

The SDK is provided as a Java Library. Your platform will need to be able to host this library. Other configurations may work such as Linux OS or Mac, 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.

You'll need to include the 'bcprov-jdk15to18-164.jar' jar in your project. It can be found in your Java Mine download folder, alongside the 'MineLibrary.jar' jar.

Sample projects have been built using Netbeans IDE 18.

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 Import statements to the top of your code file. Some of the imports we've included may not be needed for your implementation.

Code Example:

```
import com.monnit.mine.BaseApplication.ExtensionMethods.Extensions;
import com.monnit.mine.MonnitMineAPI.Gateway;
import com.monnit.mine.MonnitMineAPI.MineServer;
import com.monnit.mine.MonnitMineAPI.Sensor;
import com.monnit.mine.MonnitMineAPI.enums.eFirmwareGeneration;
import com.monnit.mine.MonnitMineAPI.enums.eGatewayType;
import com.monnit.mine.MonnitMineAPI.enums.eMineListenerProtocol;
import com.monnit.mine.MonnitMineAPI.enums.eSensorApplication;
```

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. Lastly, select the port, documentation will use the default port of 3000.

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

Code Example:

```
MineServer _Server = new MineServer(eMineListenerProtocol.TCPAndUDP, IPAddress.getByName("127.0.0.1"),
3000);
_Server.StartServer();
```

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

Adding Processing Handlers

The MineServer object will utilize processing handlers when certain things happen. For instance, when a gateway delivers a message, the "GatewayMessageHandler" handler is fired. If the message contains sensor data then additionally the "SensorMessageHandler" handler is fired. There is no need to subscribe to all handlers, just the ones you want to handle.

Code Example:

```
_Server.addGatewayDataProcessingHandler(new GatewayMessageHandler());
_Server.addSensorDataProcessingHandler(new SensorMessageHandler());
_Server.addExceptionProcessingHandler(new ExceptionHandler());
_Server.addPersistSensorHandler(new PersistMessageHandler());
_Server.addPersistGatewayHandler(new PersistGatewayHandler());
_Server.addUnknownGatewayHandler(new UnknownGatewayHandler());
```

Registering Gateways

Now that your server is operating and is ready to deliver your data, you need to register your gateway(s) so that 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.

To register a gateway with the server you need to instantiate an instance of the Gateway class. These values are best stored in a non-volatile data cache. Then it can be retrieved when the server starts and updated while the server is running if you are going to adjust the gateway parameters. This allows the values to be up to date each time the server starts.

If you have stored the rest of the gateway configurations you can apply them directly using an alternate constructor. Once the gateway object is created you can register it with the server.

Code Example:

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

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

Registering 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 instantiating a sensor object then registering it to the server. 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 server then completes a two-step registration. First, the sensor object is registered with the server, and then it is assigned to one of the registered gateways. This tells the server 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 register the same sensor object with additional GatewayIDs. 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");
_Server.RegisterSensor(12345, MineSensor);
```

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 handlers you find that the server will fire is “PersistGatewayHandler”. After information in the gateway object is updated this handler fires and allows you to store this new information into your non-volatile data store for use, or for you to use with subsequent instantiations. There is a similar handler “PersistSensorHandler” 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 “PersistGatewayHandler” handler you will see that it sends the Gateway object. By using the “FindGateway” method you can obtain the instance of the gateway object you registered with the server.

Code Example:

```
@Override
public void ProcessPersistGateway(Gateway gateway)
{
     GUIListenerFunctions.print("Gateway " + gateway.GatewayID + " has been updated");
}
```

The lookup will return the same instance of the gateway you registered. If you have overridden the Gateway class for custom functionality then your overridden class will be returned to you.

Stopping the Server

The server also has a “StopServer” method which will close and dispose of the sockets. This does not remove the registration of the gateways and sensors. If you would like to remove a gateway or sensor there are methods on the server class that enable you to do that. Removing a gateway will also remove the link of assigned sensors but will not remove the sensor object from the server allowing it to remain attached to gateways it has been assigned to. Removing a sensor will remove its assignment link from all gateways it was assigned to. (*Note: Does not affect the gateways flash memory, only the server 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.

This is important because if you had 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 eApplicationProfileType.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:

```
public static void updateSensor(long sensorID)
{
     Sensor sensor = GUIListenerFunctions.FindSensorBySensorID(sensorID);
     double hb = Double.ParseDouble(hbtxtbx.Text);
     double ahb = Double.ParseDouble(ashtxtbx.Text);
     int aphb = Integer.ParseInt(aphtxtbx.Text);
     double min = Double.ParseDouble(minthreshtxtbx.Text);
     double max = Double.ParseDouble(maxthreshtxtbx.Text);
     double hyst = Double.ParseDouble(hysttxtbx.Text);
     int failedtrans = Double.ParseDouble(ftblmtxtbx.Text);

     try
     {
          TemperatureBase.SensorEdit(sensor, hb, ahb, aphb, min, max, hyst, failedtrans);
     }
     catch
     {
          throw new Exception("Update not implemented for " + sensor.MonnitApplication.toString());
     }
}
```

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.java' file, I would recommend copying the 'UpdateTemperatureSensor.java' file. Then modifying it to fit the parameters needs of the ButtonLedBase.SensorEdit() method.

Code Example:

```
public static void updateSensor(long sensorID)
{
     Sensor sensor = GUIListenerFunctions.FindSensorBySensorID(sensorID);
     double heartbeat = Double.ParseDouble(hbtxtbx.Text);
     double awareHeartBeat = Double.ParseDouble(ashtxtbx.Text);
     int enterAwareState = Integer.ParseInt(easwli.Text);
     int rearmTime = Integer.ParseInt(rearmTime.Text);
     int failedtrans = Integer.ParseInt(ftblmtxtbx.Text);

     try
     {
          ButtonLedBase.SensorEdit(sensor, heartbeat, awareHeartBeat, enterAwareState, rearmTime,
                                   failedtrans);
     }
     catch
     {
          throw new Exception("Update not implemented for " + sensor.MonnitApplication.toString());
     }
}
```

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;
ServerInstance.RegisterSensor(SensorID, MineSensor);
```

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.