Monnit Mine
Getting Started

Monnit Corporation

Version 1.0.1.0

Contents

Introduction

Monnit Mine API was created to allow you to easily incorporate Monnit wireless sensors and gateways into your existing applications.

Requirements

At the current release, the API 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.

Sample projects have been built using Visual Studio 2012 and can be compiled and run using the Free Express version available from Microsoft here: http://www.microsoft.com/en-us/download/details.aspx?id=34673

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 API. Your best option for API 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 API questions may take longer than sensor or gateway related questions.

Basic Usage

The following will outline the basic use of the API 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 MonnitMineAPI 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.) 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. The constructor is documented here: http://mine.imonnit.com/v1.0.1.0/APIDocumentation/?topic=html/c7916769-0f6a-baf2-b191-7148dee491c9.htm

Code Example:

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

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. You can find the events documented here: http://mine.imonnit.com/v1.0.1.0/APIDocumentation/?topic=html/709a65d7-4767-17d3-1e42-b0887d73b331.htm

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, SensorMessagesEventArgs e) {}

void Handle_PersistSensor(object sender, HandlePersistSensorEventArgs e) {}

void Handle_PersistGateway(object sender, HandlePersistGatewayEventArgs e) {}

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

Registering Gateways

Now that your server is operating and is ready to deliver your event 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. Information you will need is documented here: http://mine.imonnit.com/v1.0.1.0/APIDocumentation/?topic=html/d048bbe0-4023-5c00-59b3-fab396e9946b.htm

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);

ServerInstance.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. The basic constructor for the sensor is documented here: http://mine.imonnit.com/v1.0.1.0/APIDocumentation/?topic=html/c31cd03e-65a5-e764-a1b4-72af6b7ca332.htm

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");

ServerInstance.RegisterSensor(12345, MineSensor);
ServerInstance.RegisterSensor(12346, MineSensor);
```

Other Usage Notes

The basic usage will get you up and running with the Monnit Mine API. 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 non-volatile 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 only the GatewayID. By using the “FindGateway” method you can obtain the instance of the gateway object you registered with the server.

Code Example:

```
void Handle_PersistGateway(object sender, HandlePersistGatewayEventArgs e)
{
    Gateway GatewayObj = ServerInstance.FindGateway(e.GatewayID);
    if (GatewayObj != null)
    {
…
    //Code to persist gateway to data store
…
    }
}
```

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 other gateway 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.