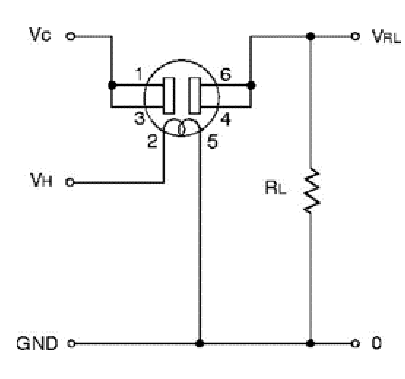
SenPro System Documentation

By Henkjan Braaksma

# Circuit

To start, let’s look at the circuit of the system. Most sensors we have, and I think, most commercial sensors out there, have two inputs and two outputs. Some have 6 pins, but two pairs of pins will be functionally identical. In this case, the identical pins can be soldered together, or one pair can be ignored. Here is an example of a TGS-822’s circuit layout, which has been functionally identical for most sensors.



As you can see, the sensor has 6 pins, but 1 and 3 are pretty much the same thing, as are 4 and 6.

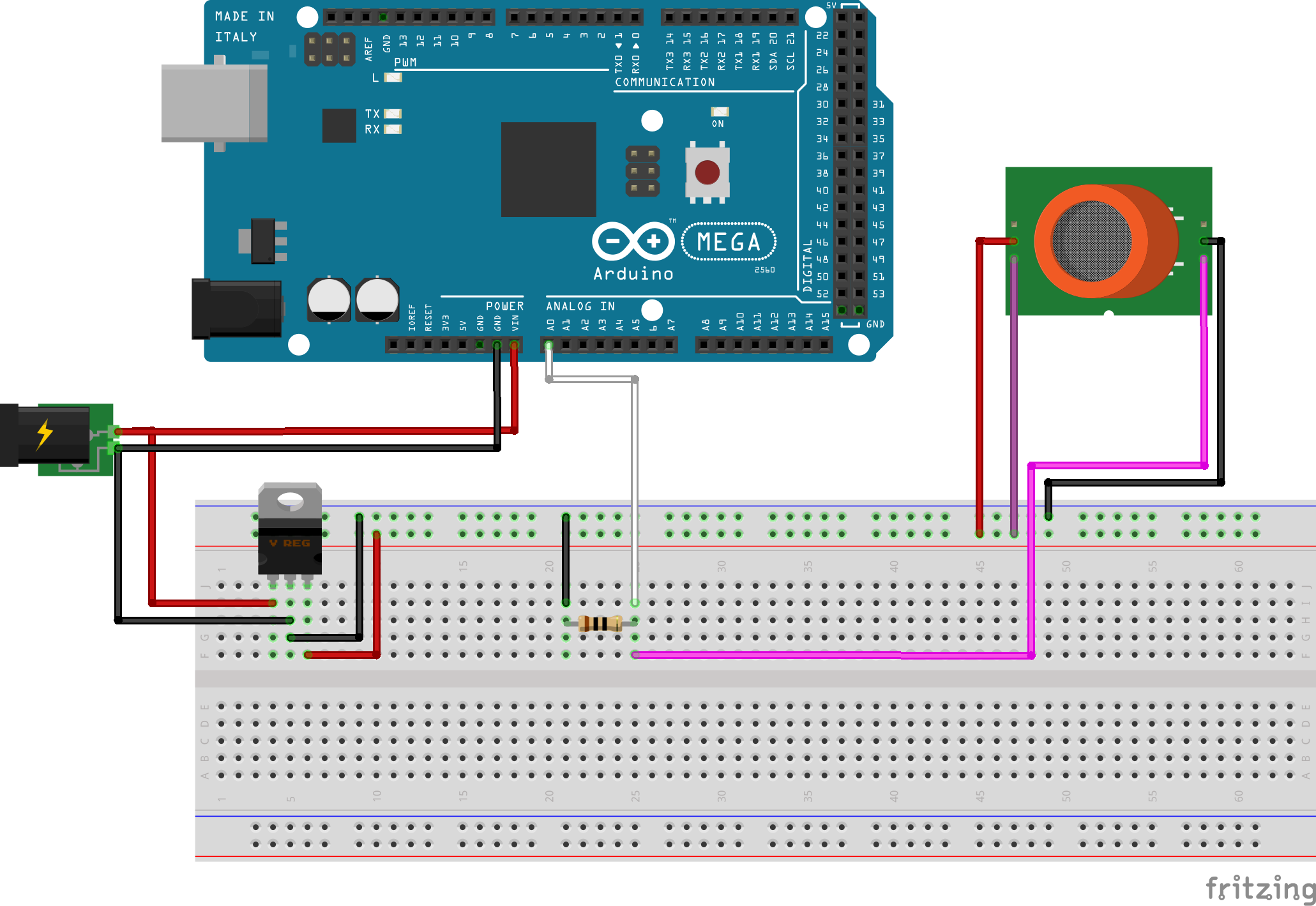
In fact, they go both ways as well: 1\3 can work as outputs as well as inputs, and vice versa for 4\6. For the sake of documentation, we will use pin 1 as input and pin 6 as output.

Two currents go into the sensor: Vc and Vh. Both currents should be 5 volts.

Vh is very simple: It’s for the sensor’s heater. It goes into the sensor, and then right out into ground. The sensor will heat up, a process that takes a few minutes.

Vc is is the signal current. The sensor is actually nothing but a big resistor. The signal current goes in, through the resistor, and then out again. The output voltage is the thing that you can use to measure the gas presence. The higher the output voltage, the higher the presence of the gases it measures.

After Vc leaves the sensor, it gets connected with the Arduino in parallel with a 10k resistor.



Above is an example of a breadboard circuit using the TGS-822 sensor. The external power source is 9 volts. It goes directly to the Arduino (which needs 9 volts to operate) and to an L7805CV voltage regulator to convert the current to 5 volts before it goes into the sensor.

This circuit has been replicated onto an Arduino prototype shield several times. See the earlier prototypes.

# Arduino

The Arduino-setup consists of a little more than just the circuit. At the base is an Arduino Mega. On top of that is a Redbearlabs BLE shield, and on top of that is a prototype shield with the sensor wiring.

This setup is not completely ideal: the Redbearlabs shield is designed for Arduino Unos, and as such covers up a fair number of pins, making them unusable. However, mounting the BLE shield on top of the protoshield causes the BLE shield to not work, unfortunately.

## Preparing the IDE

Before the code can compile on the Arduino IDE, some libraries need to be installed. Here’s a quick overview:

|  |  |
| --- | --- |
| RBL\_nRF8001 | Redbearlabs’ library for Bluetooth Low Energy. |
| BLE SDK for Arduino | RBL\_nRF8001 is dependant on this library. |
| TimerOne | Makes it easy to use the hardware timer in the C code. |
| perceptron.h | I made this library to make it a little easier to build a neural network in the Arduino code. |

Installing the libraries is easiest done like this:

1. Navigate to the Library Manager in the Arduino IDE by going to Sketch > Include Library > Manage Libraries.
2. Enter “8001” in the search bar.
3. From here you can easily find and install RBL\_nRF8001 and BLE SDK.
4. Enter “TimerOne”.
5. Install TimerOne.
6. Leave the manager and add the PerceptronLibrary.zip by going to Sketch > Include Library > Add .ZIP Library

Even if you do not use the Arduino IDE, this way you can at least download workable versions to then access using your IDE of choice.

With the libraries installed into the IDE, we can look at the code.

## Sensor Code



### 1

Inclusion of required libraries. SPI and EEPROM are included as RBL is also dependant on those, but these don’t need to be downloaded/installed, the IDE comes with them. boards.h is from the BLE SDK.

### 2

Initiates the Bluetooth Low Energy shield. The name is important: the android application looks for this specific name when trying to connect.

### 3

Sets up hardware timer 1 to fire an interrupt function every second (or every million nanoseconds). After that it attaches which function to run as the interrupt.

### 4

Begin serial output, for delicious debugging.

### 5

The ble\_do\_events() function encompasses everything the BLE shield does in every instance of the loop.

### 6

The main meat of the code is this interrupt function . Even so, it’s quite simple. The Arduino reads off the different analog pin inputs, puts them in a string and sends the string away as bytes over BLE to any recipients.

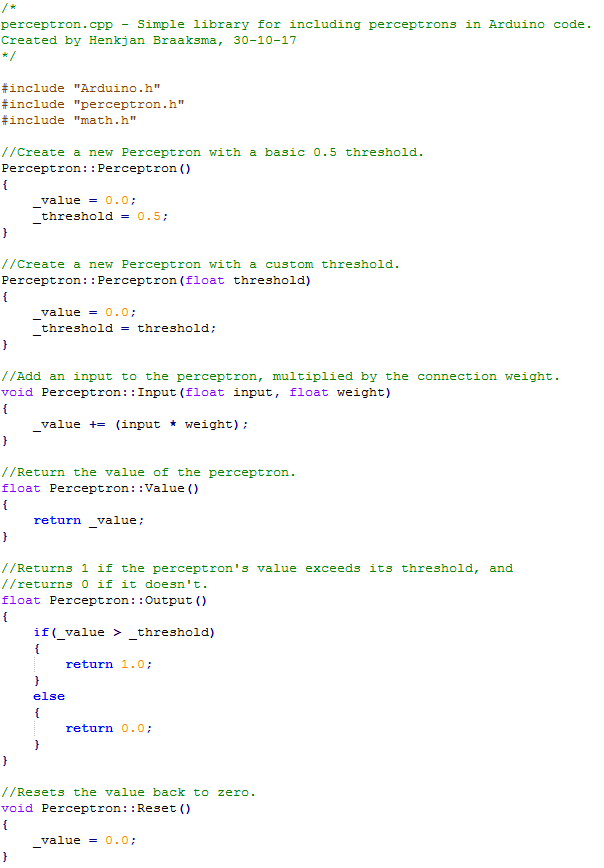
The string is formatted as a collection of numbers divided by commas. This can be easily parsed by the Android application.

## 

## 

## perceptron.h

perceptron.h is a very simple library made for the purpose of facilitating the creation of neural networks in C code for Arduino. The library consists of a single class, Perceptron. Here is its entire code.

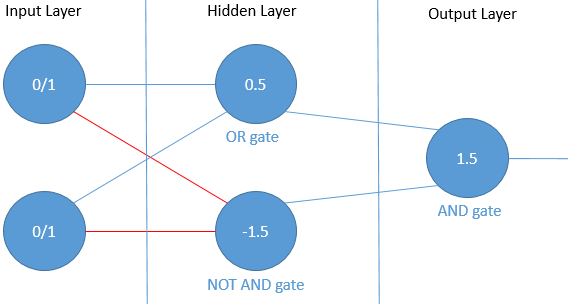


#### Usage

The code itself is simple enough, but you may be wondering how exactly to use it. I will demonstrate how to use the library to create a basic implementation of a XOR-gate.

There are 2 inputs. They produce either 1 or 0. Like every XOR gate, the intent is to output 0 (false) if the two inputs are the same, and 1 (true) if they are not.

Below is a representation of a XOR-gate using perceptrons.



The numbers in the circles of the Hidden and Output layer represent their thresholds.

The lines represent the connections between perceptrons, with their color indicating the weight: blue lines have a weight of 1, and red lines have a weight of -1.

The OR gate’s threshold is 0.5. Because the weight of its connections are 1, the inputs it gets from the input layer are either 0x1 = 0 or 1x1 = 1.

As such, it needs at least one of the inputs to be 1 in order to pass its threshold. After all, 1 > 0.5.

The NAND gate’s threshold is -1.5, and its connections have a weight of -1. This means that the inputs are either 0x-1 = 0, or 1x-1 = -1.

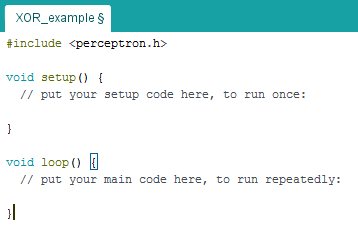
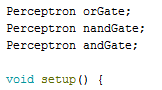
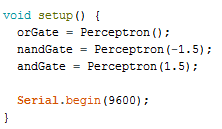
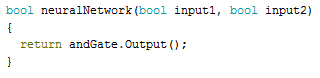
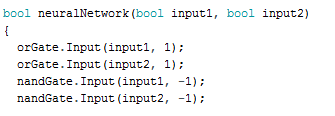
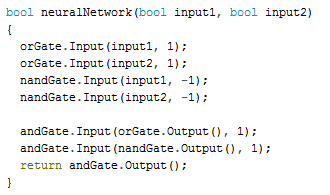
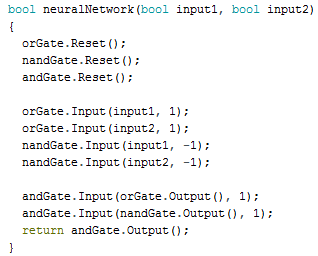
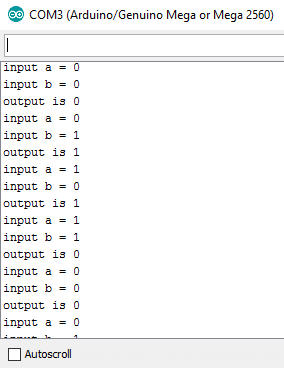
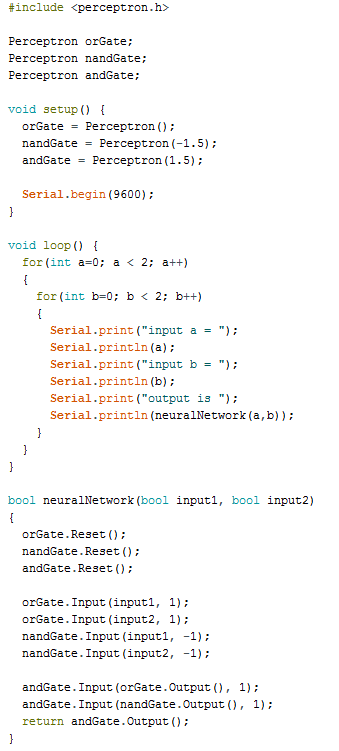
If the inputs are both 0, the NAND gate passes the threshold. 0 > -1.5.  
However, if the inputs are both 1, the threshold is not passed: the weights turn the 1s to -1s, making the perceptron’s value -2. -2 < 1.5.

Finally, the outputs of the OR and NAND gate go through a seperate AND gate in the output layer. The weights are 1, and the threshold is 1.5. So both the OR gate and the NAND gate need to output positively to get a positive output from this gate.

So the different outputs of this gate are as follows:

|  |  |  |
| --- | --- | --- |
| Input 1 | Input 2 | Output |
| 0 | 0 | 0 (fails the OR gate threshold) |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 (fails the NAND gate threshold) |

Now that we have a clear understanding of the gate, let’s write it for Arduino.

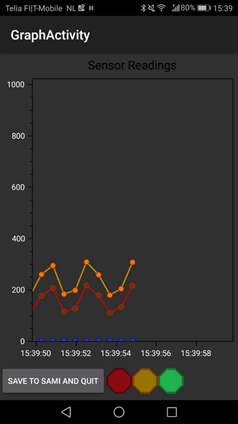
1. Start a new Arduino sketch. Add an include for perceptron.h.  
   
2. Define the different perceptrons you will be using, before even the setup and loop functions.  
   
3. Inside the Setup function, properly create the perceptron instances and give their different thresholds.   
   Note how orGate doesn’t get a defined threshold: this means it will default to 0.5.  
   Finally, write a Serial.begin for debugging.  
   
4. Let’s define the actual XOR-gate as its own function. Below the loop, create a new function. It should take two parameters, which I am making booleans here (after all, the only inputs meant for our XOR-gate are 0 and 1.  
   The output that we want to get from this function is the result from the output layer. In this case, that is simply the output of the AND-gate, so write that in.  
   
5. Next up is the actual programming of the network. With this library, a perceptron’s input can be easily modified by a single method call, and its output is also a single method call away.  
   First off, we send the inputs to the two hidden layer perceptrons.  
     
   The first parameter is the value of the input. The second parameter is the weight.
6. The next part of our function is handing the different outputs of the hidden layer to the AND-gate in the output layer. The function now looks like this:  
   
7. One last important addition to the function is this: at the start of the function, call upon the different perceptrons to be reset. Otherwise, every time the function is called, the value will still have the inputs from previous calls.  
   The completed function now looks like this.  
   
8. Finally, inside the loop function, write a program that iterates through the different possibilities. Also write some serial code so we can see the different inputs and outputs.   
   As this doesn’t pertain to the Perceptron code itself, I won’t go into detail here. The final code is below. To the right, you can read the code’s output.  
     
   

As you can see, the output is a working XOR-gate. In this case, every perceptron had a threshold and a 0/1 output, but if you want a perceptron to output its entire value, you can easily do by calling the Value function.

## Android

### Using the Application

When the application starts, first the app starts to a blank screen to check if all permissions are in order; if not, the user needs to give permission. These permissions relate to the use of internet and bluetooth capabilities of the phone. Only if the user gives their permission can the application properly function.

After giving their permission, users are greeted by the startup activity. See figure 6 for a screencap of this activity’s layout. This is a simple screen which offers the user two options: to scan for and connect to the Arduino, to subscribe to the notifications it sends out. The second option only enables once actual connection has been established.

Once the application starts listening to the Arduino’s notifications, the application switches to the Graph activity. Se figure 7. This activity forms the main part of the application. Every time the Arduino writes a new sensor reading string (which is every second), the activity takes it and displays it on a graph. The graph is constantly updating in real-time, so a user can always look over the most recent sensor readings. At the same time, it adds the data to a package of measurements. Every minute, these measurements are sent to the Savonia Measurements online database.

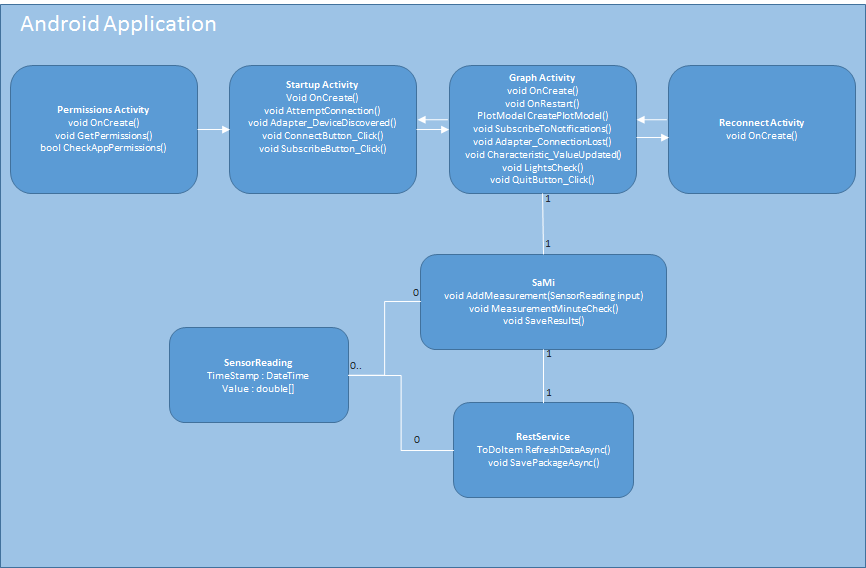
Additionally, the activity contains a save and quit button, to ensure that before shutting down the application, the last measurements are sent to SaMi, and a stoplight interface added for the purpose of demonstrating our future intent to visiting clients.

Finally, in the event that the application loses connection with the Arduino (perhaps a power outage occurred, or someone accidentally pulled the plug on the Arduino), the Graph activity automatically switches to the Reconnect activity, which will continuously attempt to reconnect to the same Arduino until it succeeds. This ensures that any use of the application can continue without problems once the Arduino becomes available to connect to again.

#### Code Overview

Writing code for an Android application means thinking in Activities. Every screen in an Android application draws its code from a single activity. Each activity can be considered their own independent application, with ways to transfer data from one activity to the other.

Considering this, making a straightforward class diagram like I did with my Windows Forms application does not quite hit the mark: although one activity can certainly start up another, this cannot be considered the same thing as a class instantiating another class. Keeping that in mind, this is a slight variant on the regular class diagram to present a clear overview of the code while taking the relation between different activities in account.



Like described earlier, the system contains 4 activities: Permission, Startup, Graph and Reconnect. Most of these do not employ any other classes themselves, except for Graph. There are no instances of data transferring from one activity to the other, with one exception: between Graph and Reconnect, the Device ID of the connected Bluetooth device is transferred to Reconnect, so it knows what ID to try and connect to.

Every activity consists of at least 4 methods:

* OnCreate, which runs when the activity is first started up.
* OnContinue, which runs continuously while the activity is active.
* OnRestart, which runs when the activity is unpaused after having been paused (by the Home button or another activity, for example)
* OnDestroy, which runs when the activity is shut down.

The descriptions of the activities that follow will only feature these methods if they are overridden in my code.

### Permission

Permission is the Main activity, which runs when the application is started up. Once running, it check with the phone if the required permissions to run the app are granted. The permissions requested are the basic permissions needed to establish a Bluetooth connection: ”Access Coarse Location”, ”Access Fine Location”, ”Bluetooth” and ”Bluetooth Admin”.

If the permissions have been granted previously, it creates and runs the Startup activity. If not, it will ask for the permissions to be granted. If this is denied, it shuts down. If this is granted, it continues to Startup.

* **OnCreate()** is overridden to show the proper message on the screen, followed by the execution of GetPermissions.
* **GetPermissions()** is a simple method: It runs CheckAppPermissions. If it returns true, it starts the Startup activity. If it returns false, it offers the user a chance to grant permissions, before checking again. If it still fails, it shuts down.
* **CheckAppPermissions().** Another very simple method: it checks with the Android Permission Manager whether permissions for using this app have been granted.

### Startup

The Startup method runs no special code automatically. All of its useful features are tied to the two buttons. The code for executing Bluetooth-related actions is supported by a NuGet Package: Plugin.BLE. This package gives the programmer access to easily usable BLE functions. Connections are handled by a BLE adapter object designed as a singleton.

* **OnCreate()** is overridden for basic means: setting up the correct layout and tying the UI elements to the code.
* **AttemptConnection()** sets up the search for the Arduino: it sets Adapter\_DeviceDiscovered as the event handler for when the BLE adapter discovers a device, then instructs the adapter to start scanning for BLE devices.
* **Adapter\_DeviceDiscovered()** fires as soon as the adapter discovers a device. The device doesn’t instantly move to connect. First, it checks if the device’s name is “SenProBLE”. Only when the name matches does it try to connect. If it connects succesfully, it displays as much on the screen, and enables the Subscribe button to become clickable.
* **ConnectButton\_Click()** runs the AttemptConnection function.
* **SubscribeButton\_Click()** starts the Graph activity.

### SenProReading

SenProReading is the successor to the Measurement class of the first prototype and fulfills the same function: It is a container class used by Graph and SaMi to facilitate the delivery of a sensor reading. It consists of a DateTime-format timestamp and an array of double values representing the different sensor readings.

### Graph

The Graph activity, as described before, is the central point of the application, and where most of the work happens. From here, the messages created by the BLE Shield are parsed to arrays of doubles, which then get added to both a live-updating graph and a package of measurements that are sent out to SaMi once per minute. The graph is handled by the OxyPlot plugin. OxyPlot is a C# library with functionality for many different types of applications, including Android. Not unlike LiveCharts that was used during the first prototype, it makes the implementation of a graph in the application quite easy.

* **OnCreate()** is overridden for mostly basic means. It sets up the layout, ties the layout elements to the code, instantiates a SaMi object, runs CreatePlotModel to set up the Oxyplot graph and finally executes SubscribeToNotifications, which sets the whole activity in motion. It also stores the device ID of the BLE device currently connected.
* **OnRestart()** is overridden with one main goal: to check if the application is currently listening to the Android notifications (it stops listening if it loses connection) and, if not, to resubscribe to notifications.
* **CreatePlotModel()** sets up all the basic necessities for the Oxyplot plot element in the layout. This includes creating the axes, and creating the LineSeries which will be turned into lists of separate points on the graph: one LineSeries per sensor. When displayed on the graph, all points of a LineSeries are connected, creating a line graph.
* **SubscribeToNotifications()** attempts to start listening to the notifications the Arduino produces. BLE devices are divided in Services, which are in turn divided in Characteristics. To subscribe, the application must navigate to the correct characteristic and set up an event handler to fire whenever that characteristic’s value changes. This is what SubscribeToNotifications does. The indexes of the correct service and characteristic depend on the BLE device used: for the RBL BLE-shield we are using, the service index is 2, and the characteristic index is 1.
* **Adapter\_ConnectionLost().** This is the event handler that fires whenever the phone loses connection to the Arduino device. It takes the previously stored BLE-shield ID and starts the Reconnect activity, transfering the device ID with it.
* **Characteristic\_ValueUpdated().** When the application is subscribed to the BLE-shield’s notifications, this handler will fire every time the value changes: which is every second as the Arduino creates and sends out a new string. When it fires, it takes the string, checks if it is the correct amount of numbers (by checking the amount of comma’s), parses the string to doubles, and adds the doubles to the OxyPlot LineSeries, which adds them to the graph. It also adjusts the bottom axis so the newest point always appears on the same point while the rest slowly trail out of view. After this, it also adds the measurement to the SaMi-object using SaMi’s AddMeasurement method. It also runs SaMi’s MinuteCheck, and if it returns positive, tells the SaMi object to upload the saved measurements, so the measurements are uploaded every minute. Then it runs LightsCheck. Finally, it runs Oxyplot’s InvalidatePlot method, which causes the application to update the plot, with all new additions and changes.
* **LightsCheck().** As you can see in figure 7, the layout has several images representing a stoplight. This function was added mostly as a form of demonstration: it simply checks if one of the sensors has exceeded a certain value, and changes the images of the lights accordingly: if the first sensor exceeds 600, it turns red. If it’s between 300 and 600, yellow. If below 300, green.
* **QuitButton\_Click().** Though a user can of course just shut down the app, using the Save and Quit button ensures that any unuploaded measurements for SaMi are uploaded to SaMi before quitting.

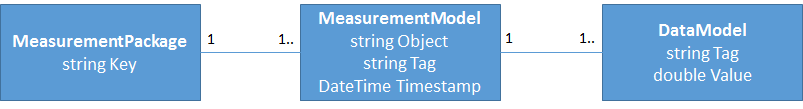
### 

### 

### SaMi

The SaMi class supports the Graph activity by handling any actions regarding the storage and upload of Savonia Measurements-bound data. It does not run anything by itself: all methods need to be called by its parent class (the Graph activity being the only class that instantiates a SaMi object).

The SaMi functionality, for the most part, was already written by Savonia developers. C# projects can add a reference to SaMi to include this functionality to their project. The basic workings of SaMi is based off the following hierarchy:



**MeasurementPackage** is the object that the application uploads. When uploading to SaMi’s database, a package’s key is verified: different projects get different keys, and if the key is invalid, the data won’t get placed in SaMi.  
A MeasurementPackage can contain a number of **MeasurementModels**. Models are groups of measurements of the same project made in the same timeframe.

A MeasurementModel, in turn, consists of a number of **DataModels**. A DataModel is a singular measurement, or in my case, a singular reading from one sensor.

Because this hierarchy is slightly abstract, here is an example of the package as the application uploads it. Every minute, the SaMi class uploads a single MeasurementPackage with the project’s write key. The Package contains 60 MeasurementModels, one for every second of the minute. The Object string is “Halax Prototype” and the Tag is “Gas Sensors”. Inside every MeasurementModel are 4 DataModels, one for every sensor. The Tag is the sensor’s identity (Sensor 1 through 4) and the Value is the sensor’s reading.

Most of the SaMi class is dedicated to storing the different readings into this hierarchy, and then uploading the resulting MeasurementPackage, after which the package is cleared and the process begins anew.

* **SaMi()** is the class’s constructor. It instantiates a couple of things: the Rest Service object, a list object for MeasurementModels and the package variable, complete with write key.
* **AddMeasurement(SenProReading readingInput)** AddMeasurement takes the readingInput parameter and then saves it in a way fitting the SaMi hierarchy: It creates a new MeasurementModel using readingInput’s DateTime, and then stores every value in readingInput’s Value array in its own seperate DataModel. Once this is done, the MeasurementModel gets added to the list object.
* **MeasurementMinuteCheck()** is a simple check method: it checks if the list of MeasurementModels: if it exceeds 60 (1 for every second), it returns true, and false otherwise.
* **SaveResults()** takes the list of MeasurementModels that has been built up and stores it in the package variable. Then, using the Rest Service method SavePackageAsync, attempts to upload it to the SaMi site.

### RestService

Uploading the package data goes through an HTTP POST method which is contained in the RestService class. Where the SaMi class prepares the data and puts it in the right format to be sent to the online database, the RestService performs the actual upload. Because HTTP Post only be used to upload strings, the RestService also converts the entire object to a Json string using the Newtonsoft Json library. Once arrived at the database, the system knows to convert it back to its respective objects, and from there includes it in the database.

* **RestService().** The constructor creates an HTTP client that the other methods can work with.
* **SavePackageAsync(MeasurementPackage item, bool isNewItem).** First, this method takes the item, and serializes it and everything it contains into a Json string. if isNewItem is true, it then tries to post the Json to the SaMi server, awaiting a success status code. If it fails at any time, the exception is caught and written to console for debugging.

### Reconnect

The Reconnect activity exists as a failsafe to ensure the application can keep working even when connection to the Arduino is temporarily lost. It consists of only a single method, the override of the OnCreate method.

**OnCreate()** is overridden to set up the layout and display a message that connection is lost and reconnection is being attempted. Then, it reads the device ID that was transferred to it by the Graph activity on startup, and uses the Bluetooth adapter to continuously attempt to reconnect to the device owning that ID. On a success, it closes itself, returning the user to the Graph activity, which then restarts automatically.