Internet of Things

Project Report

Submitted To:

Prof. Dr. Anna Förster

Dr. Ing. Andreas Könsgen

Idrees Zaman

Submitted By:

Abdullah Shahwani	3063021
Mina Foosherian	3061773
Jibin P. John	3061574

Table of Contents

Preface

The explosive growth of the "Internet of Things" is changing our world and the rapid drop in price for typical **IoT** components is allowing people to innovate new designs and products at home. The internet of things is developing now because we've figured out how to give everything we produce an address, we have enough bandwidth to allow device-to-device communications, and we have the capacity to store all the data, those exchanges create. This was the motivation to carry out a project in this domain using the already developed and tested platform of "**Mole Net**", an underground wireless sensor network created and maintained by the ***ComNets*** department at the ***University of Bremen***, Germany.

We are very thankful to Prof. Dr. Förster to offer us such an opportunity where we get to learn about and experiment with Arduino, Internet of Things, Underground Sensor Networks, and Wireless Communication; all at once.

Introduction

The aim of the project is to wirelessly transmit the data (observations) from the buried underground sensor nodes to a hand held device. The **MoleNet** sensor nodes communicate at *433MHz* frequency. An immediate device is designed, in our case, Arduino Uno (with Atmega328p controller) that receives the data at 433MHz and tunnels the data to the connected hand held device (for instance, a smart phone). The received data is then logged and visualized on an Android application.

We took a simple and straight-forward approach to carry out this task; by using minimum hardware, that is, cost-effective, power-efficient and easily available.

Arduino (Uno)

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. Thanks to its simplicity and accessible user experience, Arduino has been used in thousands of different projects and applications. The Arduino software is both easy-to-use for beginners, and flexible enough for advanced users. It runs equally well on Mac, Linux, and Windows.

We picked ***Arduino Uno*** for our project, as this is the simplest Arduino board, frequently available; it has a lot of support already present over the internet and it has a USB port, through which one can easily provide connectivity to other interfacing devices.

The Clock speed of Arduino Uno is 16 MHz so it can perform a particular task faster than other processors or controllers. Following is a labelled diagram of a typical Arduino Uno board:

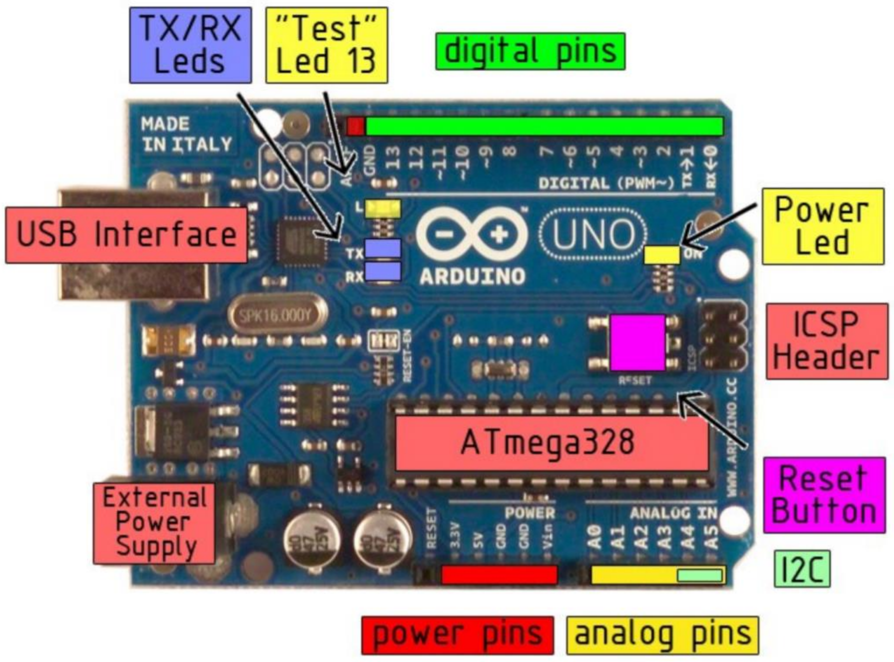


Figure 1. Arduino Uno explained!

Hardware Design

As defined earlier, the aim of the project is to wirelessly transmit the data from the buried underground sensor node to a handheld device (as portrayed in the figure below).

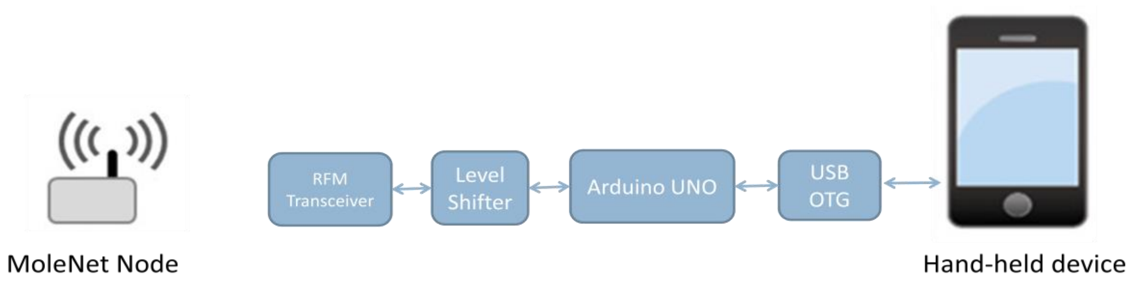


Figure 2. Block diagram of the project

In order to reach our goal, we used the following set of hardware:

Element	Description
RFM69 transceiver (433MHz)	It receives data from buried Gateway (MoleNet); and works at the same frequency as the underground node
Arduino Uno (with Atmega328p 5v 16MHz)	Acts as a translator between smart-phone and buried node; to take the data as input and forward to the hand held device for storage and visualization
Logic Level Converter (3.3v ~ 5v)	It steps up 3.3V signals to 5V and steps down 5V into 3.3V, in order for the Arduino board and RFM69 transceiver to communicate
USB OTG cable	Connects Arduino Uno board to the smart-phone

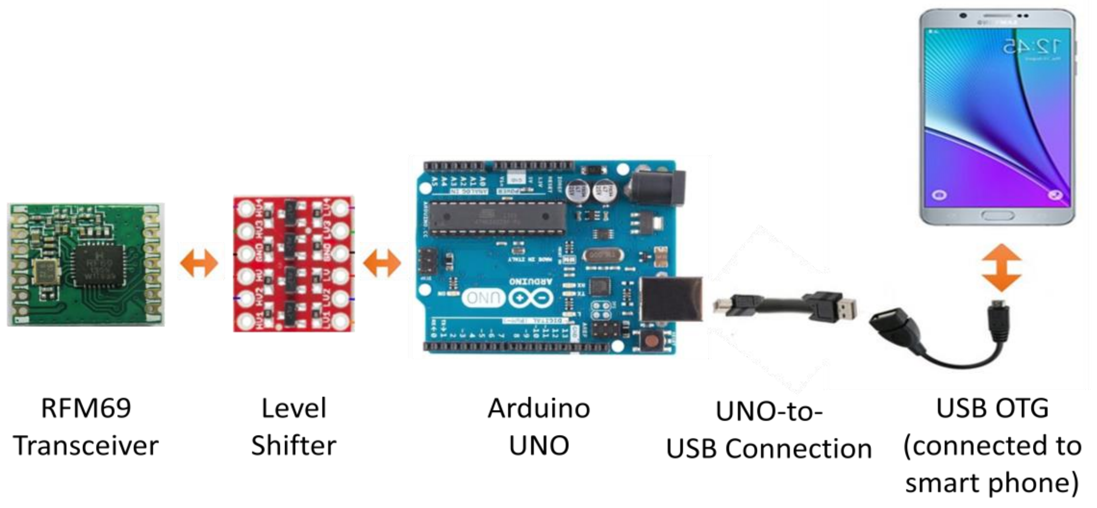


Figure 3. Hardware used

Software Implementation

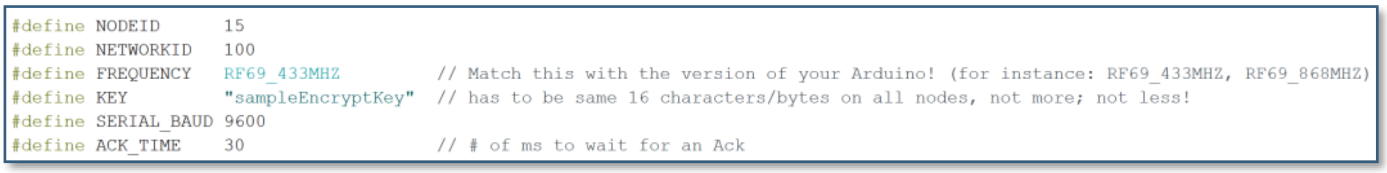
We used the Arduino Software (IDE) that contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus. It connects to the Arduino hardware to upload programs and communicate with them. We wrote the code on Arduino 1.8.2 desktop application. The latest version of the code can be obtained from Github:

https://github.com/jibinpjohn/Molnet_IOT/

Following are the libraries we used and the functions they perform:

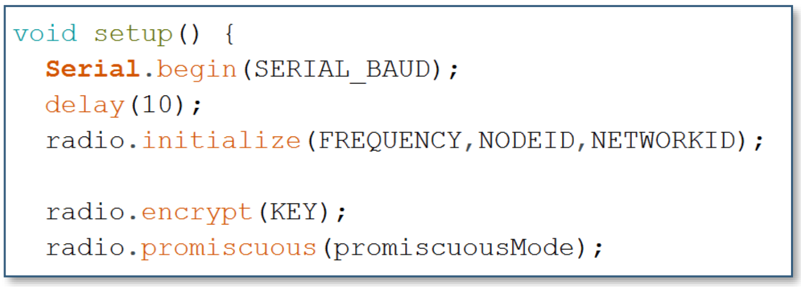
Library	Description
RFM69.h	Initialized an object **radio** to receive the data at the transceiver; also used ***promiscuousMode*** to sniff any other data that might be present on the network
SPI.h	Used to print received data on the serial port
LowPower.h	Used **Sleep** functionality to save energy; Uno will always be in sleep until explicitly awakened

Although the entire code can be viewed at the above provided link, here are some excerpts from the Arduino code:



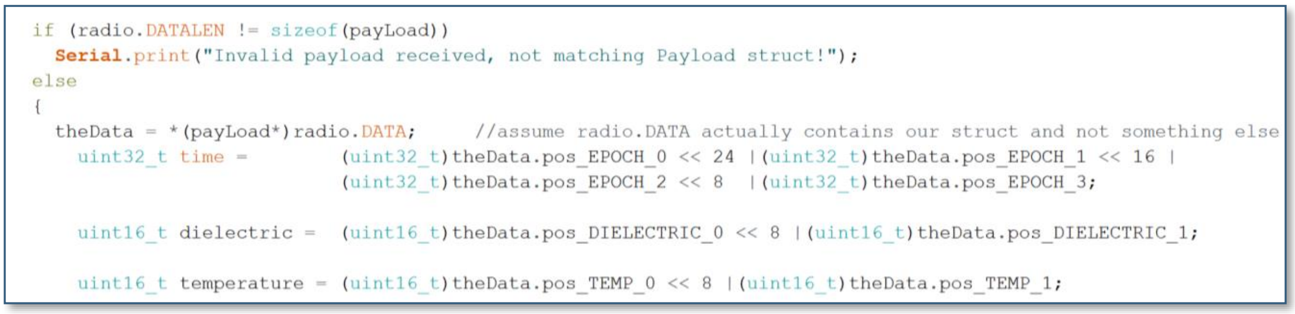
```
#define NODEID      15
#define NETWORKID   100
#define FREQUENCY   RF69_433MHZ          // Match this with the version of your Arduino! (for instance: RF69_433MHZ, RF69_868MHZ)
#define KEY         "sampleEncryptKey"   // has to be same 16 characters/bytes on all nodes, not more; not less!
#define SERIAL_BAUD 9600
#define ACK_TIME    30                   // # of ms to wait for an Ack
```

In the Figure above, some global parameters are defined so they can be easily called whenever required.



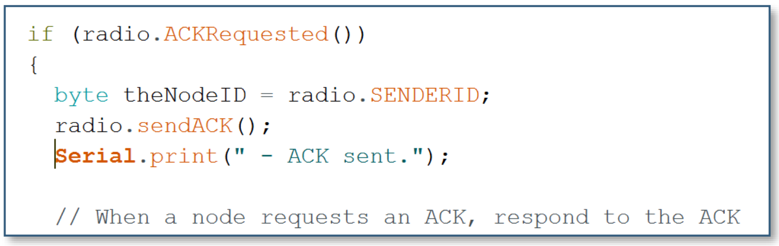
```
void setup() {
  Serial.begin(SERIAL_BAUD);
  delay(10);
  radio.initialize(FREQUENCY,NODEID,NETWORKID);

  radio.encrypt(KEY);
  radio.promiscuous(promiscuousMode);
```

In the Figure above, one can observe the use of ***encryption*** and ***promiscuous*** methods to add security in the program.



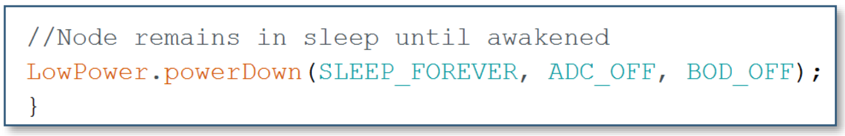
```
if (radio.DATALEN != sizeof(payLoad))
  Serial.print("Invalid payload received, not matching Payload struct!");
else
{
  theData = *(payLoad*)radio.DATA;      //assume radio.DATA actually contains our struct and not something else
    uint32_t time =         (uint32_t)theData.pos_EPOCH_0 << 24 |(uint32_t)theData.pos_EPOCH_1 << 16 |
                            (uint32_t)theData.pos_EPOCH_2 << 8  |(uint32_t)theData.pos_EPOCH_3;

    uint16_t dielectric =  (uint16_t)theData.pos_DIELECTRIC_0 << 8 |(uint16_t)theData.pos_DIELECTRIC_1;

    uint16_t temperature = (uint16_t)theData.pos_TEMP_0 << 8 |(uint16_t)theData.pos_TEMP_1;
```

We also provided check for any incomplete or erroneous packets. If such a packet is encountered, it will be ignored and an error message will appear on the screen. Otherwise, the program will execute normally.



```
if (radio.ACKRequested())
{
  byte theNodeID = radio.SENDERID;
  radio.sendACK();
  Serial.print(" - ACK sent.");

  // When a node requests an ACK, respond to the ACK
```

When a node requests for an Acknowledgement, it will be provided through the ***sendACK()*** method.



```
//Node remains in sleep until awakened
LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
}
```

Most importantly, in order to conserve energy, the Arduino always remain in ***Sleep***, until explicitly interrupted at the hardware. This way we were able to significantly cut down on the energy consumption.

Power Consumption

Since RFM69 Transceiver has to remain ON during entire time so that no data can be missed; the only way energy can be saved is if we make the Arduino ***sleep*** until there is some significant data at the antenna. For that we used **LowPower** library (link below), to turn OFF Uno microcontroller along with the Analog-to-Digital Converter (ADC) and Brownout Detector (BOD) modules.

https://github.com/LowPowerLab/LowPower

The table below shows how much Power is conserved through this approach:

Component	Mode	Power Consumption	Duty Cycle
Arduino Uno	ON	46.5 mA	-
	Sleep	31.4 mA	100%
RFM69 Transceiver	Receive Mode	~16 mA	100%
	Sleep	1 μA	-

Android Application

Android is very popular mobile operating system developed by Google. It is estimated that around 1 Billion devices are operated in android operating system. It is based on Linux kernel. In our project we decided to develop an android application for data visualization. Our app has following basic features:

- Broad cast receiver, immediately detects when the device is attached
- App has device filter, to check vendor ID before granting permission
- It stores the data to a file.
- The graphs get updated in real-time.

Application uses following libraries or **API** for different purposes:

- USB connection Management by felHR85's **UsbSerial**.
- **MPAndroidChart** library for plotting Graphs.

The Android studio has been chosen as application development environment, which can be downloaded from the internet free-of-cost. The following section contains details about the android application development and the necessary files.

Main Components

There are 3 major files in an Android application (described below). There are a host of other files too but they are all linked together with the help of these main files.

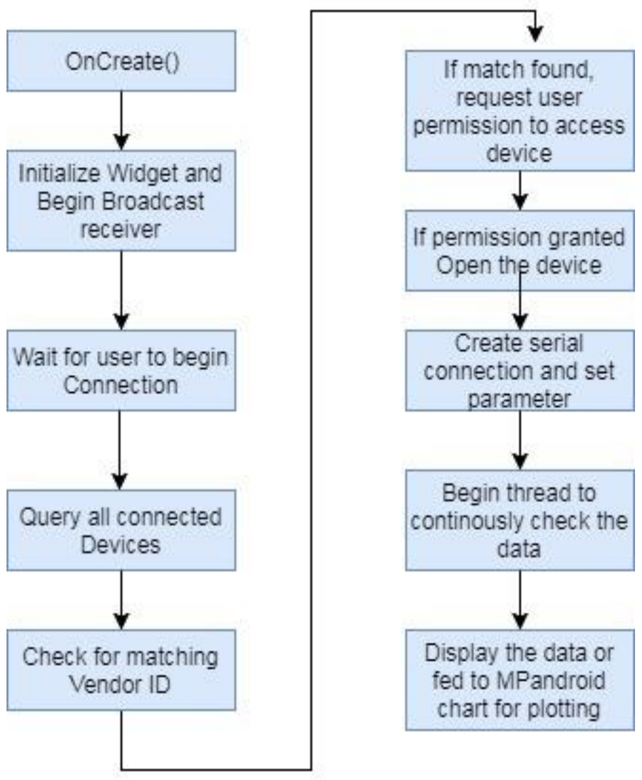
File	Function
Activity.java	This is where the Java code goes. It controls the way the app will function.
activity_main.xml	This contains the layout of the app, i.e. the components or the widget-like buttons, TextViews etc.
AndroidManifest.xml	This is where you define when the app must start, which permissions it needs, and which hardware it needs to access.

An ***activity*** can be described as a screen where the user interacts with the phone. Activities contain ***widgets*** like buttons, text fields, images, etc., which help in the transfer of information. We used one activity, the **Main Activity**, which will take the user's input to send to the Arduino and also display the received text.

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: **onCreate()**, **onStart()**, **onResume()**, **onPause()**, **onStop()**, and **onDestroy()**. The system invokes each of these callbacks as an activity enters a new state. The operations during these stages can be coded into the ***Activity.java*** files.

Program Flow

Below is a simplified program flow of the Android application we developed.



OnCreate()
Initialize Widget and Begin Broadcast receiver
Wait for user to begin Connection
Query all connected Devices
Check for matching Vendor ID
If match found, request user permission to access device
If permission granted Open the device
Create serial connection and set parameter
Begin thread to continously check the data
Display the data or fed to MPandroid chart for plotting

Figure 4. Program Flow Diagram

The USB Serial Library

It is a tedious task to set up a serial connection in Android. Among all the relevant libraries, we found **felHR85** as a good one from ***Github***. In order to add this library to our project, we downloaded latest **JAR** file from Github.

Opening a Connection and Receiving the data

A ***BroadcastReceiver*** is defined in order to receive the broadcast .It asks for user permission and also to start the connection automatically when a device is connected. It closes the connection when it is disconnected. If the user has granted permission, initiate a connection for the device whose vendor ID matched our required vendor ID. Also, if a broadcast for a device attach or detach is received, manually call the methods for the Start and Stop buttons. A ***SerialPort*** is defined using the device as the connection as the arguments. If this is successful, open the SerialPort and set the parameters accordingly.
What extra permissions the app might require is stated in *manifest file*. The following line is added to manifest file for the permission to make the phone a USB host.

```
<uses-feature android:name="android.hardware.usb.host" />
```

The app can be made to start automatically by adding an ***IntentFilter*** to the MainActivity. This IntentFilter will be triggered when any new device is attached. The kind of device can be explicitly specified by providing the vendor ID and/or product ID in an XML file.

```
<meta-data
            android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
             android:resource="@xml/device_filter" />
    </activity
```

For receiving the data, a reference of a Callback is passed to the read function so that it will automatically trigger when any incoming data is detected. We had already inserted some delimiters at the beginning and end of the packet in the adruino codes which served as a reference point to application, to understand beginning and end of a packet and makes extracting the data easier. Also every bytes in 24 byte packet also separated by delimiters, this allows the app to distinguish each parameters separately. The following code has been used to extract the data.

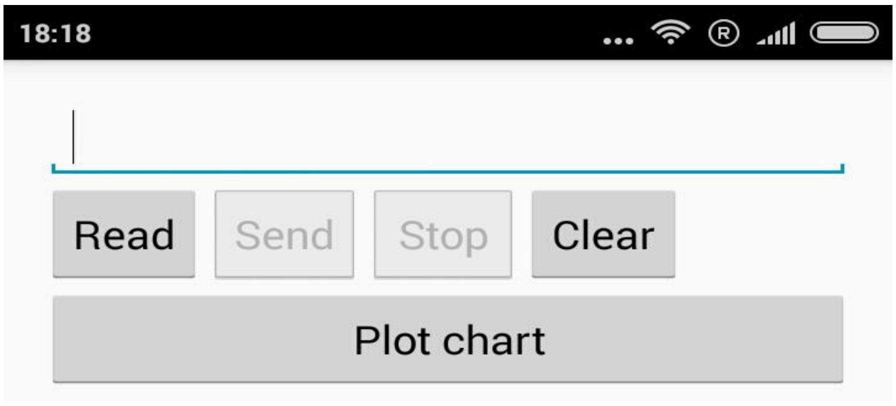
```
UsbSerialInterface.UsbReadCallback mCallback = new UsbSerialInterface.UsbReadCallback() { //Defining a Callback which triggers whenever data is read.

    @Override
```

```
public void onReceivedData(byte[] arg0) {
  String data = null;
  byte[] buffer = new byte[256];
  try {
    data = new String(arg0, "UTF-8");
    buffer=(byte[]) arg0;
    String strIncom = new String(buffer, 0);
    sb.append(strIncom);
    int endOfLineIndex = sb.indexOf("/CS");
    int startCS = sb.indexOf("CS");

    if (endOfLineIndex > 0) {
      String sbprint = sb.substring((startCS + 2), endOfLineIndex);
      Date date=new Date();
      DateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
      DateFormat format2 = new SimpleDateFormat("HH:mm:ss");
      // String formatted = format.format(date);

      String formatted2 = format2.format(date);
      sample.setTime_stamp(formatted2);
      String segments[]=sbprint.split("/");
      int decimal=Integer.parseInt(segments[2]);
      sample.setTemperature((float)decimal);
      decimal=Integer.parseInt(segments[1]);
      sample.setDielectric((float)decimal);
      addEntry();
      addEntry2();          }
```

The part of code which indicated in pink used to determine the start and end of the packet. The part of code indicated in yellow extract the bytes based on the relative position. This data is saved into a variable in a class and **addEntry()** used to put points into the graph.

The Plot Activity

To navigate from main activity to another activity, one has to click the ***plot chart*** button. We extracted only the necessary data that is, dielectric and temperature values.



We fed this data to the plot instantaneously without storing it anywhere (for initial testing purpose). ***MPAndroidChart API*** is used for visualizing the data. Details about this ***API*** can be found here:

https://github.com/PhilJay/MPAndroidChart

The graph has following functionalities:

- The graphs get updated in real time.
- Plot has zoom functionality.
- Data being pushed to left after specified entries.

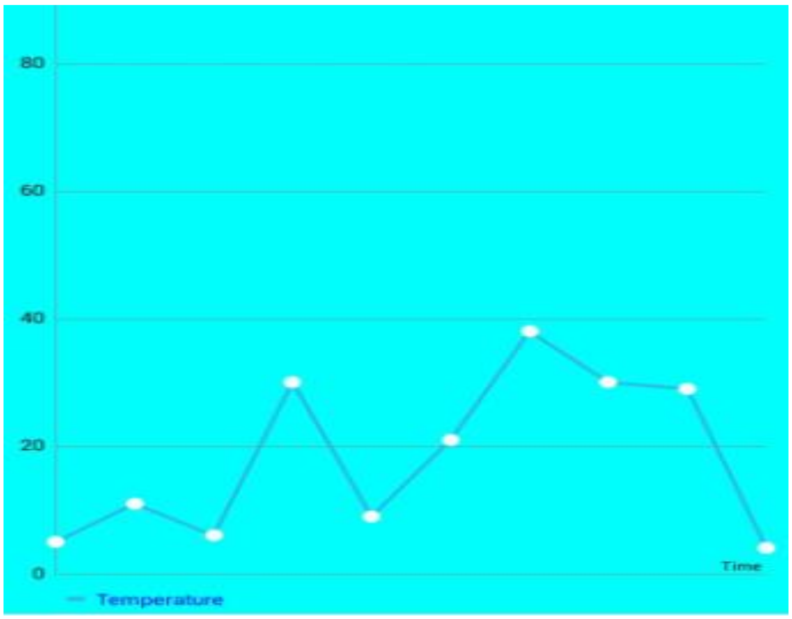


Figure 5. A snapshot of temperature data being plotted real time

As mentioned before, every time serial data is read the application calls **addEntry()** function to add data to graph. The following code gives a bird view of how the data is being fed to the graph in real time, however there are lot of configurations related to graph are required in order to function this.

```
public  void addEntry() {
    LineData data = mChart.getData();
    if (data != null) {
        ILineDataSet set = data.getDataSetByIndex(0);
        // set.addEntry(...); // can be called as well

        if (set == null) {
            set = createSet();
            data.addDataSet(set);
        }
        data.addEntry(new Entry(set.getEntryCount(),sample.getTemperature()), 0);
        data.notifyDataChanged();
        // let the chart know it's data has changed
          mChart.notifyDataSetChanged();
        // limit the number of visible entries
          mChart.setVisibleXRangeMaximum(30);
    }
}
```

The data is being taken from the class variable and added to the graph. Some api related functions being called to notify the graph that data is changed.

Future Enhancement

Following amendments are possible to further improve this simplistic design. We believe that the following immediate changes can be easily designed and implemented:

- Further data sets can be added to the graph based on the ***nodeID*** of transmitting sensor node, so that more data can be obtained and analysed simultaneously
- A database can be incorporated to store all the received data for future use and any analysis or tests.
- A separate interface can also be included to retrieve and display the stored data values (for instance temperature and dielectric).

Apart from these obvious enhancements, there is a lot of room for many other improvements, as both Arduino and Android are open-source and has a great level of flexibility.

References

- https://github.com/LowPowerLab/RFM69/
- https://learn.sparkfun.com/tutorials/rfm69hcw-hookup-guide
- https://www.allaboutcircuits.com/projects/communicate-with-your-arduino-through-android/
- https://github.com/ComNets-Bremen/WUSN