THE UNIVERSITY OF BRITISH COLUMBIA

Department of Electrical and Computer Engineering

PROJECT 2

LoRa Water Quality Monitor

Prepared by

Lab Group #: 6

Young Hoon Ji (36400166)
Craig Bonamis (20474145)
Ammar Rehan (10649151)
Tonny Li (19313048)

In Partial Fulfillment of the Requirements for

ELEC 391 - Electrical Engineering Design Studio II

Date Performed: August 2nd, 2019

Date Submitted: August 14th, 2019

ABSTRACT

We implemented a water quality monitor using an Arduino MKRWAN 1300 microcontroller board, turbidity sensor, water temperature sensor, and LoRa communication module. LoRa was used to allow communication between our microcontroller and a University of British Columbia gateway. We chose to pursue this project because there have been reports that certain UBC buildings, including MacLeod, provide subpar water quality. Thus, the design objectives of this product are: (1) to provide a convenient way for the user to monitor the water quality of a certain building or area and (2) have the ability to easily compare the water quality of multiple buildings or places through an interactive map. This would allow both UBC students and staff to monitor the water quality of different buildings on campus. Since we only had one microcontroller to work with, we focused on monitoring the water quality of the MacLeod building for this project. We accomplished our project goals by uploading turbidity and temperature data to The Things Network via LoRa WAN wireless communication. From there, the data is sent to Tago, which is an Internet of Things web application platform for visualizing data. An LCD Screen and button were added to allow the user to take our project to multiple buildings to test out their respective water quality. After completing the project, we find that our implementation offers a convenient method of monitoring water quality across different locations. The only drawback of our implementation is that Tago did not allow the water quality of each sensor to be displayed on the map. A separate screen had to be navigated to for seeing the water quality associated with each sensor. We recommend more time, financial resources, and testing in order to definitively recommend this project for implementation at UBC.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

Gateway	A device used to connect two different networks. Often acts as a source of internet connection.
Turbidity	A measure of the degree to which the water loses its transparency due to the presence of suspended particulates.
Nephelometric	An apparatus that is used to measure the size and concentration of particles in a liquid by analysis of light scattered by the liquid

LIST OF ABBREVIATIONS

GPS	Global Positioning System.
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
LCD	Liquid Crystal Display
LoRa	Long Range
LoRaWAN	Low Power Wide Area Network
LPP	Low Power Payload
NTU	Nephelometric Turbidity Unit
TTN	The Things Network
UBC	University of British Columbia
WHO	World Health Organization

INTRODUCTION

This project report presents an investigation of a water quality monitor that can be used by UBC students and staff to monitor the water quality across different campus buildings. The objectives of this report are to describe the hardware, Arduino software, and Tago front-end features used for the implementation of this project.

This report is divided into three sections:

In Section 1, the hardware of our project is explored.

In Section 2, the Arduino software is analyzed. This includes its overall control-flow.

In Section 3, the Tago web application platform is discussed. This includes the drawbacks and benefits of this platform in comparison to the first platform we tested, Cayenne.

Finally, we draw conclusions and offer recommendations.

1.0 PROJECT HARDWARE

1.1 Introduction

This section explores the hardware we implemented to achieve our goal of water temperature and turbidity measurement. In addition, it explores the hardware required for LoRa connectivity and the hardware that was implemented to improve the user's experience.

1.2 Design Process

Our project was centered on communicating with the LoRa gateway. LoRa communication was made possible by an antenna and LoRa communication module that came with the MKRWAN 1300 Arduino microcontroller board. The antenna was connected directly to the Arduino board. Through LoRa communication, turbidity and temperature data would be sent to a UBC gateway and then uploaded to the internet. With our design goals in mind, we bought appropriate turbidity and temperature sensors. The turbidity sensor we bought did not come with a driver board, which complicated our measurement retrieval process. After testing each sensor, we connected them as shown in Figures 1.1-1.4. We enclosed both turbidity and temperature probes in a 3-D printed structure so it would be easier to carry out measurements. The process used for measuring water turbidity and the temperature is discussed later in the project report.

We also implemented an LCD screen to show us real-time levels of turbidity and temperature as well as a button which either enabled or disabled LoRa communication. To make our hardware self-contained and organized, we 3D printed a case for our breadboard.

1.3 Results

Figure 1.1 shows a block diagram of our circuit component connections. Figure 1.2 shows our implementation of it, followed by a close-up picture of the sensors in Figure 1.3. Sample messages on the LCD screen are shown in Figure 1.4 and Figure 1.5. Proper operation of all hardware was demonstrated in the project demo.

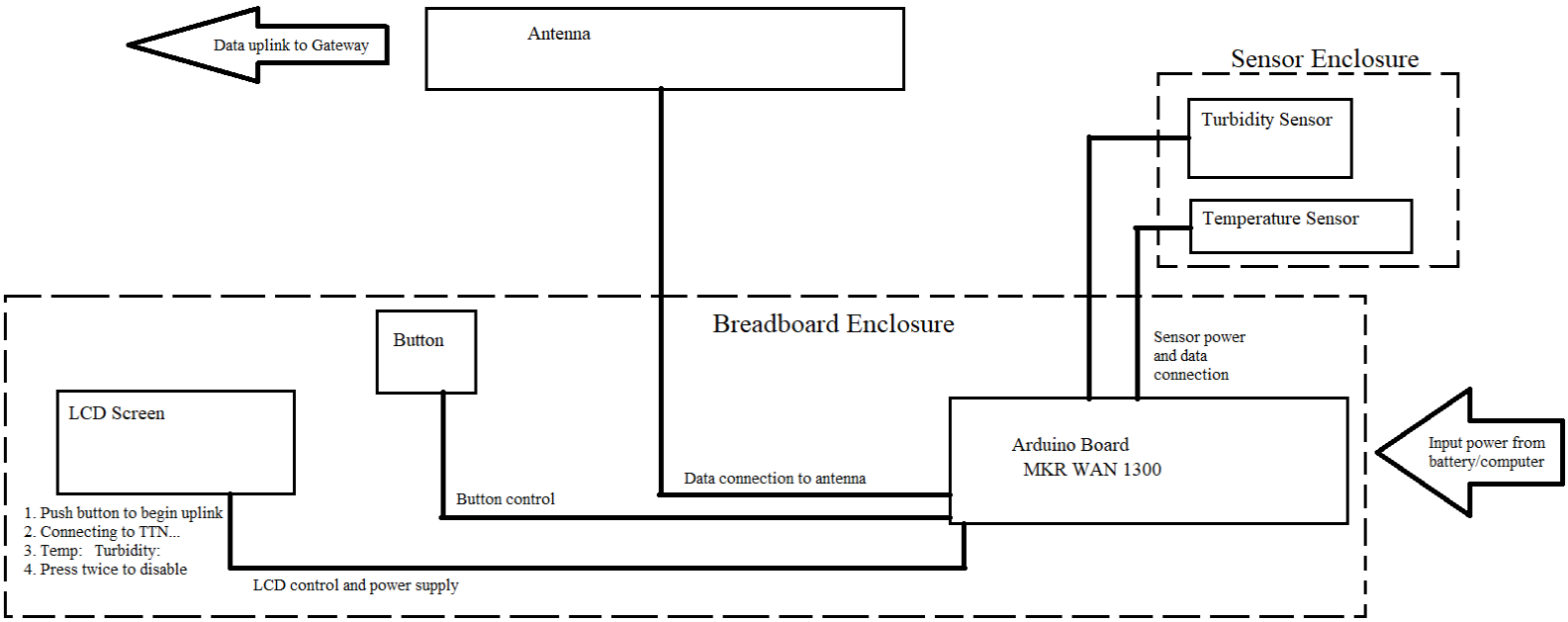

Figure 1.1 Project Block Diagram

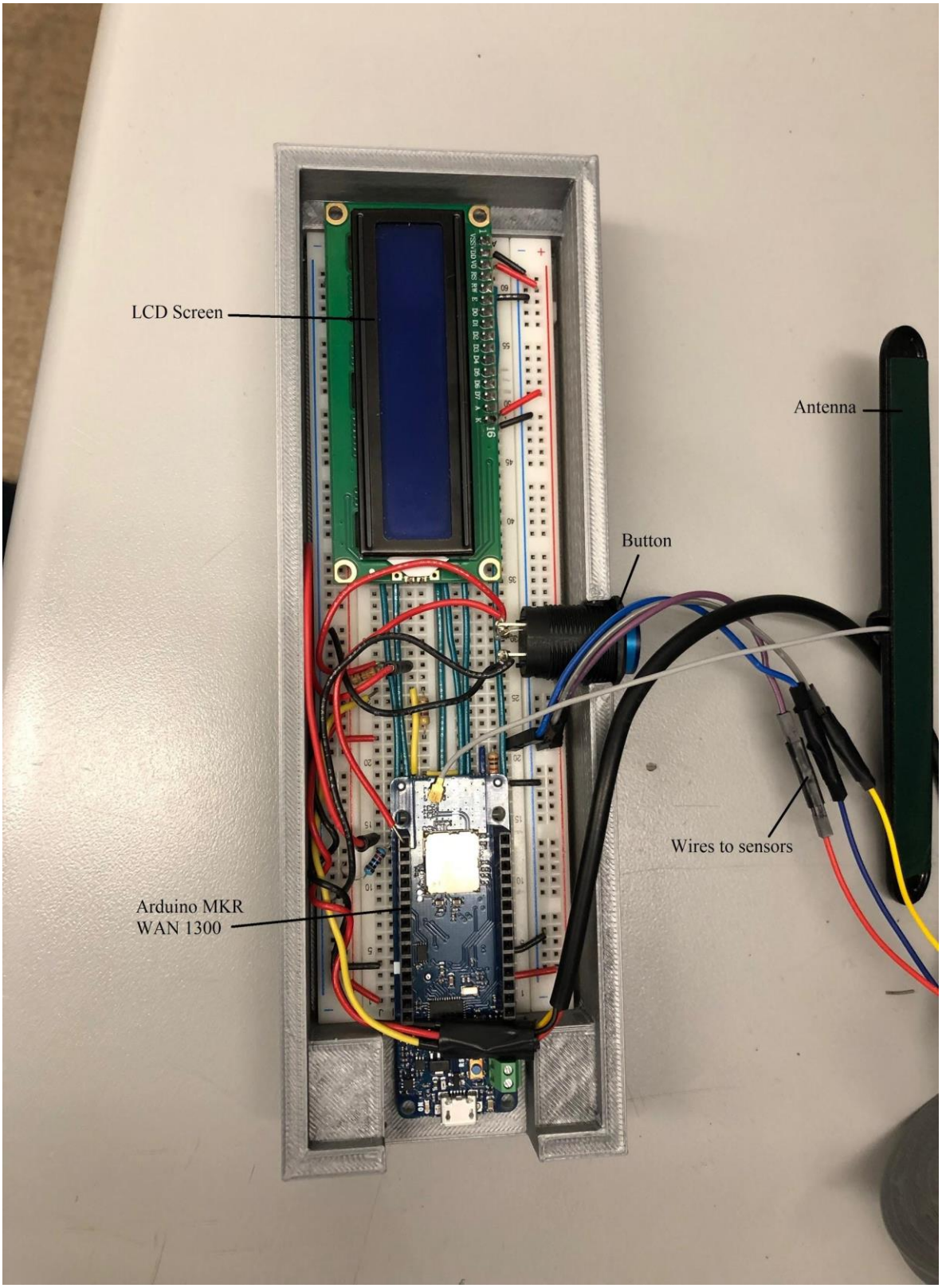

Figure 1.2 Project Implementation

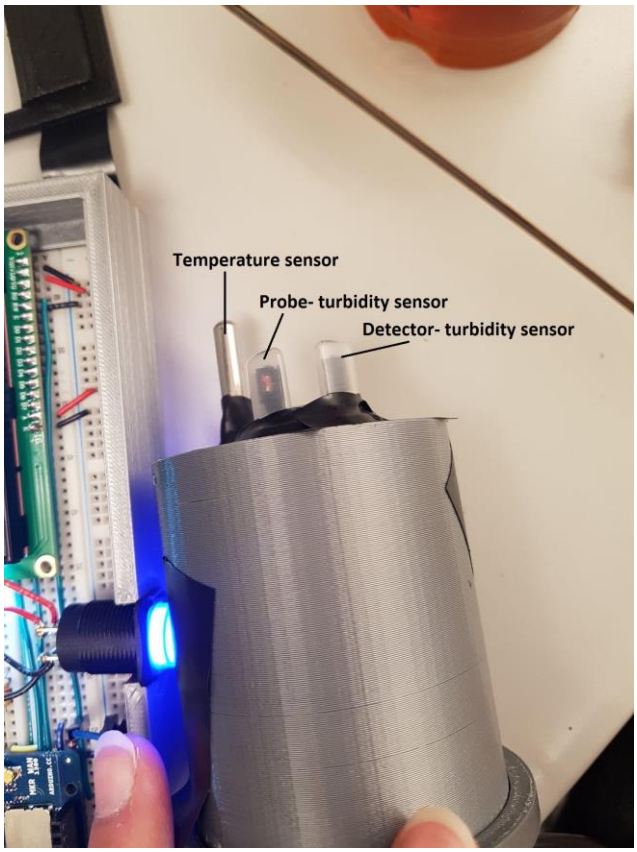

Figure 1.3 Sensor Close-up

Figure 1.4 Connecting display

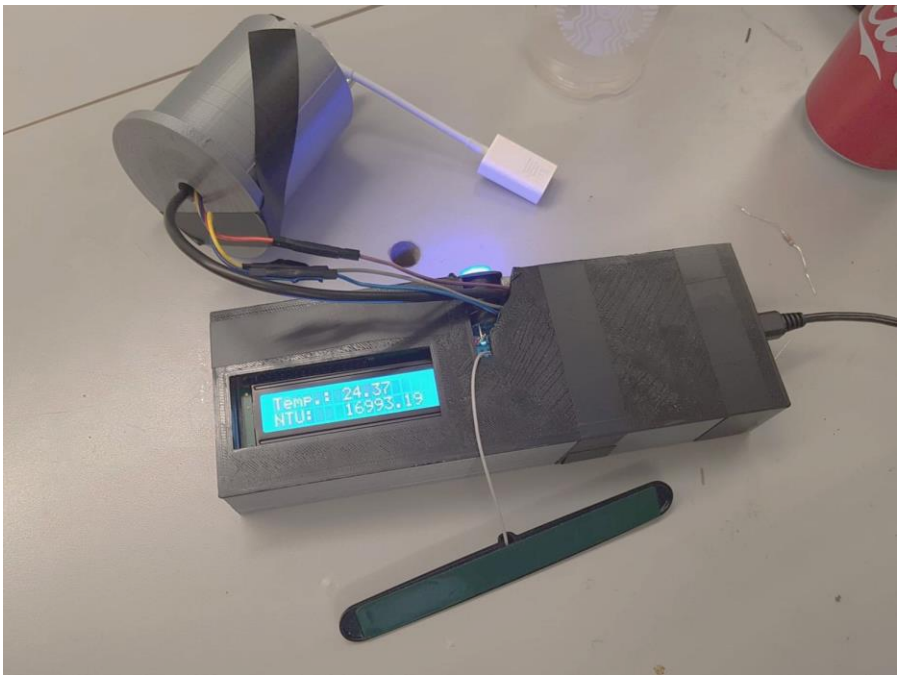

Figure 1.5 Temp/NTU display

1.4 Discussion

1.4.1 Turbidity Sensor

Turbidity is a measure of the solid particles suspended in water and is used as a metric to determine water quality. Our turbidity sensor, also known as a nephelometer, is comprised of two parts: probe and detector. The probe sends out an electromagnetic wave of low frequencies around 170 Hz [1] and a detector orthogonal to the probe detects the signal. The sensor operates on the principle of scattering since electromagnetic waves are scattered when they hit suspended particles at an angle. Nephelometers using this principle can detect particles ranging in size from 10-3 mm3 to several cm3 [6]. The units for turbidity are given in Nephelometric Turbidity Units (NTU). NTU is defined such that a suspension of one mg/l is equivalent to 3 NTU. It is largely based on Formazin Turbidity Unit (FTU), where a suspension of 1.25 mg/L hydrazine sulfate

and 12.5 mg/L hexamethylenetetramine in water has a turbidity of one FTU [4] (since mg/l in NTU are not a viable metric of measure of scattering).

For our implementation, we did not have a driver board with the turbidity sensor. Thus, we had to place a resistor in series with the current that was being produced by the turbidity sensor. The resistance was chosen so that the voltage across the resistor ranged from 0 V to 5 V, which is the voltage range that the Arduino board can handle. An analog input pin on the Arduino board was used to receive these voltage readings.

1.4.2 Temperature Sensor

We used an Arduino-compatible water temperature sensor in our project. The temperature sensor uses a thermistor. Its electrical properties change with temperature, and a corresponding voltage is produced so that the Arduino can determine the temperature of the water. It is clearly shown in Figure 1.3.

1.4.3 LCD and Button

We connected a button and LCD to the Arduino microcontroller to increase the user-friendliness of our project. The button is connected in a pull-up format to an external interrupt pin on the Arduino which constantly reads a high voltage, or a digital one. When the button is pressed, the pin reads a digital zero which prompts the Arduino to trigger an interrupt service routine. The software details of this interrupt is given in Section 2 of this report.

The LCD is there to inform the user of what is currently happening in the program flow. After the Arduino is powered on, the push of the button prompts the LCD to display the "Connecting to TTN..." message as shown in Figure 1.4. If we press the button again after this point, the connection to the gateway will be terminated and the LCD will display an appropriate message. When the system successfully connects to the TTN network, data uplink begins and that is when the sensors start taking readings. The real-time readings are displayed on the LCD screen as shown in Figure 1.5. Averaged readings are uploaded to the internet via LoRa.

1.4.4 LoRa Communication

The antenna required for LoRa communication was given to our group by the ELEC 391 course instructors. We found that LoRa connectivity was very poor. This is most likely due to technical difficulties stemming from the UBC gateway which our microcontroller was supposed to connect to. Initially, it was difficult for us to determine whether our LoRa issues were software-related or hardware-related. After consultation with the ELEC 391 course instructors and other students, we found out that the UBC gateway was the most likely culprit behind the connectivity issues. This resulted in longer wait times before the project was able to connect to the gateway, however once connected, data was consistently uploaded and visible on the internet.

2.0 ARDUINO SOFTWARE AND FLOW-CONTROL

2.1 Introduction

In this section, the arduino software for our project will be explored. This includes the overall control-flow of our project. In addition, the programming required for the LCD display and button will be analyzed.

2.2 Design Process

Our software design process began with LoRa connectivity. Since our project hinged on the upload of data to The Things Network via LoRa, we reasoned that configuring the LoRa connection to the UBC gateway should be our priority. Also, we did not have our sensors at this point as they were being shipped to us, so we could not work with them. We researched the process of connecting to TTN via LoRaWAN on the official Arduino website. Using the Arduino IDE, software was developed to link our MKRWAN1300 Arduino microcontroller with TTN. An account was set up on the TTN website and appropriate settings were changed so that a connection could be established with our microcontroller. As mentioned in Section 1 of this report, connecting to TTN via LoRa was extremely challenging due to the technical challenges posed by the UBC gateway.

After software was written to connect to TTN, we focused on capturing sensor data. The first sensor we worked with was the turbidity sensor, as it provides better information on water quality than the temperature sensor. As mentioned previously, we didn't have a driver board for the turbidity sensor so voltage readings had to be taken off a resistor using an analog input pin on

the Arduino microcontroller. Through analog-to-digital conversion, we converted the current coming out of the turbidity sensor to a voltage between 0 and 5 Volts. This voltage was then used as the independent variable in an approximation equation to yield the NTU value of whatever liquid that the sensor was reading. The approximation equation we used was derived from measuring liquids of well-established turbidity levels like Coca-Cola and milk. The software that was written to accomplish these computations is shown in Figure 2.4.

We then wrote software to capture data from the water temperature sensor. We used a pre-existing library named DallasTemperature.h to easily capture temperature data. The temperature library allowed us to use methods such as “requestTemperatures” and “GetTempCByIndex(0)” to measure temperature easily.

Both temperature and turbidity data were sent via LoRa in an LPP object. CayenneLPP.h library was used to create our payload and parse it in the TTN, thus the data could easily be attached with “ addTemperature(1, avg_temp)” and “ lpp.addAnalogOutput(2, avg_ntu)” methods.

As mentioned earlier, an LCD was used to provide the user with the ability to monitor the real-time sensory data when in front of our project hardware. A button was also provided to allow the user to start and stop the upload of sensor information to TTN. Special Arduino libraries were used for the software implementation of the LCD display and button. When integrating the button, we first thought that a timer would be the most efficient way of checking whether the button has been pressed during the operation of the rest of our program. However,

the MKRWAN1300 could not support our desired timer implementation, so we decided that the button should trigger an interrupt. Furthermore, the button would generate an interrupt in the software every time it was pressed and accordingly, the upload of data would either start or stop.

Given the complexity of our project's software, we decided to implement a state machine to manage both button functionality and main loop functionality. The switch/case programming tool was used to implement the state machine. More information on our state machine implementation is given in the "Results" section of this part of the report.

2.3 Results

Using the information in Figure 2.1, we programmed our Arduino microcontroller to connect with the UBC gateway via LoRa. In the code under Appendix A, we have declared these as a constant char pointer containing the string of application EUI and application key.

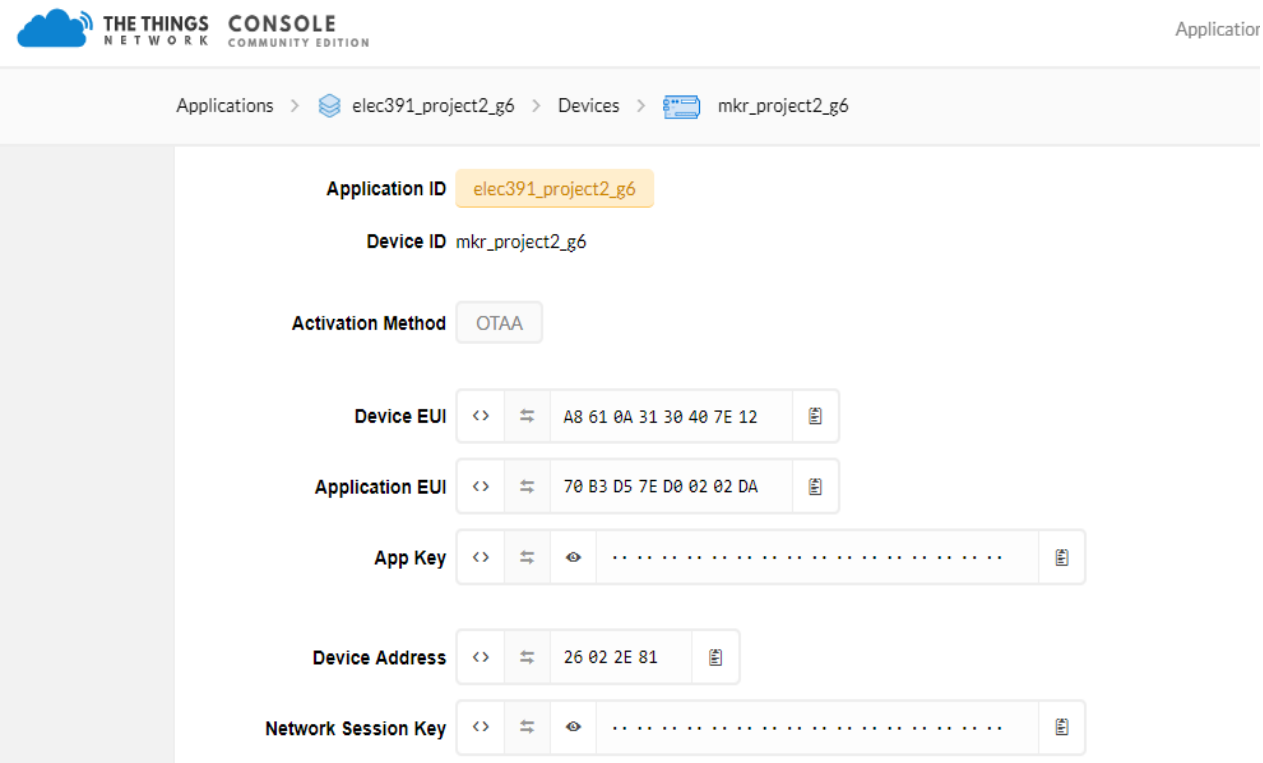

Figure 2.1 The Things Network Application Summary Screen

To connect to LoRa, we used the following code in Figure 2.2 and Figure 2.3. It makes use of pre-existing objects and methods that are specifically designed for the MKRWAN1300 Arduino microcontroller.

```
void init_lora ()
{
  Serial.println("Initializing LORA");
  while (!Serial);
  if (!modem.begin(region)) {
    Serial.println("Failed to start module");
    while (1) {}
  };
  Serial.print("Your device EUI is: ");
  Serial.println(modem.deviceEUI());
}
```

Figure 2.2 Code for Initializing LoRa Communication

```
void connect_TTN ()
{
  // Connection Loop
  while (connect_loop && (state == 1))
  {
    init_lora();
    connecting_to_TTN_msg();

    Serial.print("Connecting... \n");
    int connected = modem.joinOTAA(appEui, appKey);

    Serial.print("State while Connecting is:");
    Serial.println(state);

    if (state == 0) return;

    if (!connected) {
      Serial.println("Something went wrong; are you indoor? Move near a window and retry");
      Serial.println("Reconnecting in 1 minute");
      reconnecting_in_1_min_msg();
      delay(60*1000);
      modem.restart();
    }
    else
    {
      Serial.println("Successfully joined the network!");
      success_TTN_msg();
      connect_loop = 0;
      state = state + 1;
    }
  }

  if (state == 1)
  {
    Serial.println("Enabling ADR and setting low spreading factor");
    modem.setADR(true);
    modem.dataRate(5);
  }
}
```

Figure 2.3 Software for Connecting to The Things Network

As seen in Figure 2.4, the NTU and temperature values are calculated in the "measure_temp_volt()" function. For the turbidity sensor, an NTU value is calculated from the voltage at an analog input pin. The analog voltage is read using the "analogRead()" function. The analog voltage value is then used in an equation to calculate the corresponding NTU value. A conditional statement is also present to provide an upper and lower bound for NTU values. For the temperature sensor, the method, "getTempCByIndex" yields the temperature value.

```
void measure_temp_volt ()
{
  // Get Temperature
  Serial.println("\n-- Requesting temperatures...");
  sensors.requestTemperatures();      // Send the command to get temperatures
  temperature = sensors.getTempCByIndex(0);

  // Get current Turbidity Voltage
  NTU = (float) analogRead(Turbidity_in);
  Serial.print("ANALOG INPUT IS:");
  Serial.println(NTU);

  NTU = -20.158 * NTU + 17638.25;
  if (NTU < 0.0) NTU = 0.0;
  if (NTU > 5000.0) NTU = 5000.0;

  // Add to avg_temp, global variable
  sum_temp = sum_temp + temperature;
  // Add to avg_ntu, global variable
  sum_ntu = sum_ntu + NTU;
  Serial.print("\n----------------------------------------");
  Serial.print("\n-- Current Temp: ");
  Serial.print(temperature);
  Serial.print("\n-- Current NTU is: ");
  Serial.print(NTU);

  // LCD Printing
  display_data_msg(temperature, NTU);
  // Printing
  Serial.print("\n-- Sum Temp: ");
  Serial.print(sum_temp);
  Serial.print("\n-- Sum NTU: ");
  Serial.print(sum_ntu);
  Serial.print("\n----------------------------------------");
}
```

Figure 2.4 Function measure_temp_volt for Turbidity and Temperature

Next, the press of the button located on our project's hardware acted as an external interrupt which prompted the software to jump to the "ButtonHandler" function as an Interrupt Service Routine, or ISR, as seen in Figure 2.5.

```
void ButtonHandler ()
{
  while (digitalRead(BUTTON) == LOW)
  {
    delay(50);
  }
  Serial.println("------ Button Pressed!! ------");

  switch (state)
  {
    case 0:
    {
      if (b_flag == LOW) state = state + 1;
      break;
    }
    case 1:
    {
      state = 0;
      abort_msg();
      b_flag = HIGH;
      break;
    }
    case 2:
    {
      state = 0;
      abort_msg();
      b_flag = HIGH;
      break;
    }
    default:
    {
      state = 0;
      break;
    }
     Serial.println("------ ENd of button Handler! ------");
  }
}
```

Figure 2.5 Button ISR from Appendix A

The ButtonHandler function evaluates the current state, and either stops the upload of data or begins the upload of data to TTN. Figure 2.6 represents the flow diagram of this next-state-logic for easier visualization.

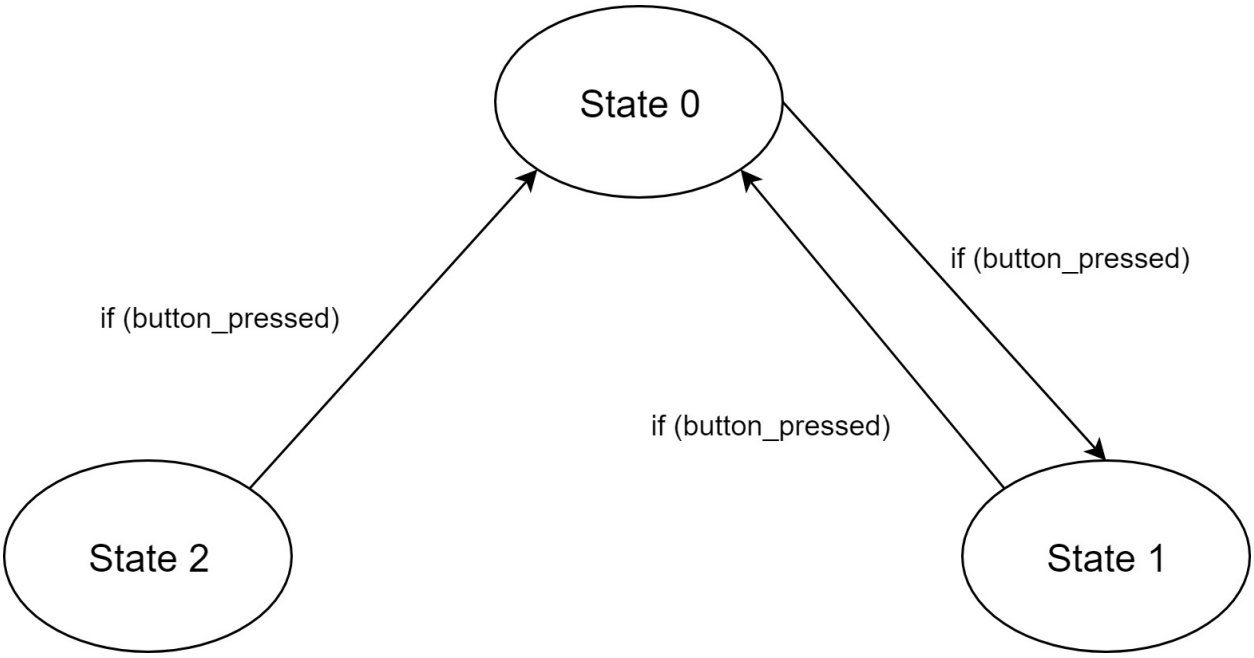

Figure 2.6 ButtonHandler Interrupt Service Routine Code

The main loop of the Arduino code also needs different functionalities for each state. In state zero, the loop waits for a button-push to begin uplink. In state one, it attempts to establish a connection with TTN before beginning uplink. In state two, data is taken, averaged, and then sent to TTN periodically. This is done until the button is pushed. This can be seen in Figure 2.7 and Figure 2.8.

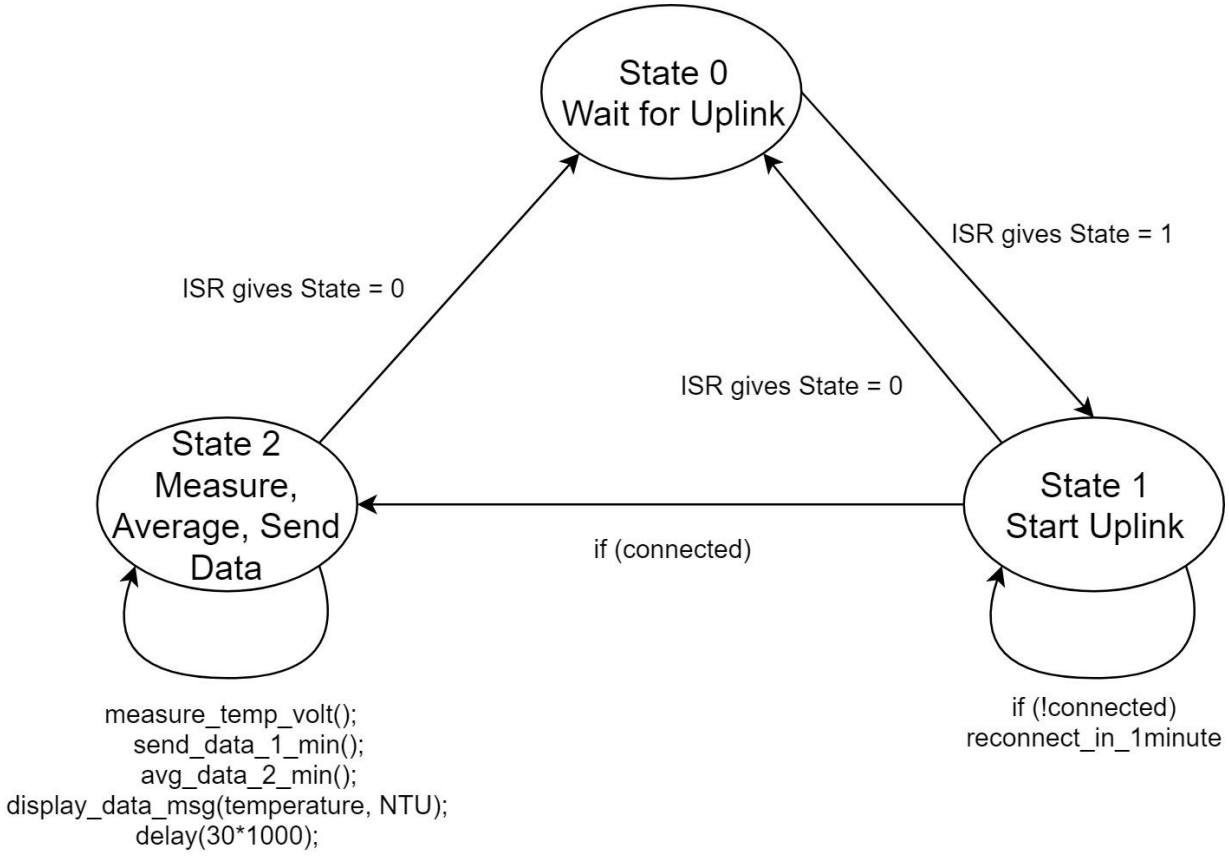

Figure 2.7 Main Loop State Machine Logic

```
void loop()
{
  switch (state)
  {
    case 0: // Wait Screen
    {
      // LCD "Push Button to Uplink" Screen
      b_flag = LOW;
      Serial.print("State is:");
      Serial.println(state);
      wait_for_uplink_msg();
      delay(1000);
      break;
    }

    case 1:
    {
      Serial.print("State is:");
      Serial.println(state);
      // Connect, check for abort
      start_uplink();
      break;
    }

    case 2:
    {
      Serial.print("State is:");
      Serial.println(state);
      // MAIN LOOP FOR 30 SECONDS
      measure_temp_volt();  // Every 30sec, measure Current Data, add it to sum
      send_data_1_min();    // Every 1 minute, Sending avg data to TTN
      avg_data_2_min();     // Every 2 mins, we average sum, update the avg variables
      display_data_msg(temperature, NTU);
      delay(30*1000);
      break;
    }

    default:
    {
      Serial.print("State is:");
      Serial.println(state);
      Serial.println("FATAL State Machine ERROR");
      state = 0;
      delay(10*1000);
      break;
    }
  }
}
```

Figure 2.8 Main Loop Software

2.4 Discussion

2.4.1 LoRa Wireless Communication Software

The software required to connect to TTN via LoRa was quite straightforward, as our code makes use of software libraries that are tried and tested. Connecting to TTN was tedious as most of the time, the software returned an error saying that a connection could not be established to TTN. Occasionally, data was sent to TTN even when the software produced a connection error. In addition, it often took a very long time for TTN to show that the data had been received. Thus, a downlink system in which TTN can immediately send a confirmation to our Arduino that it received data would be very useful for debugging purposes.

2.4.2 Sensor Software

For the temperature sensor, there weren't any issues encountered when reading the temperature data. However, for the turbidity sensor, most of the issues resulted from its hardware, as it did not seem to be very sensitive. This could possibly be explained by the absence of the driver board. It was not shipped with the sensor when we ordered it on Amazon due to time constraints. Other issues arose from initializing and summing sensor data. We decided to average sensor data from 4 measurements over a period of 2 minutes. This data was then sent every minute to TTN. Effectively, we were sending the same data twice. This was done so that we can be more confident that TTN received the data it was supposed to receive. We also noticed that the initial average of both turbidity and temperature data was 0. Thus, we created a function to initialize the average variables so garbage values do not get sent to TTN.

2.4.3 LCD and Button Software

For the LCD, we again made use of existing Arduino libraries. As seen in Appendix A, the LCD functions had "_msg" in the name to indicate an LCD display functions. These functions were made up of methods such as "clear", "setCursor(x,x)", and "print" which updated the LCD display.

For the button software, the flow diagram of the external button interrupt service routine can be seen in Figure 2.4. The main purpose of this ISR is to reset and stop the main loop as well as to return to an initial waiting state in which the user is asked to "Push Button to Begin Uplink".

2.4.4 Flow-Control of Project

The main flow diagram of the program is seen in Figure 2.5. We implemented a state machine as the functionalities of the button were different in each state. State zero initializes the uplink process while states one and two are there for stopping the uplink process. Thus, there is only a state one to state two transition in the main loop.

State two is the most important state as it serves many functions. First, state two repeats every 30 seconds until an external interrupt causes a transition to state zero. The reason for this is to measure sensor data every thirty seconds, and average it every two minutes, and then send it to TTN every one minute. We noticed that connections to TTN were not very stable and payloads were lost from time-to-time, which was why we sent the same data twice. Also, TTN could not receive more than one message packet per minute. Measurements were carried out every thirty

seconds so that users can see live data on the Arduino, thus saving floating-point calculation time for the Arduino CPU.

Our state machine implementations made it very easy to debug our project's software as we could always see the state that we are currently in when a problem occurred. Thus, we highly recommend a state machine implementation for this type of project.

3.0 TAGO FRONT-END

3.1 Introduction

We chose Tago as the web application platform for our project because it provided greater flexibility than many other free web applications that allow data to be visualized. It matched our need for an organized and convenient way to represent data.

3.2 Design Process

We first tested the web application called Cayenne and evaluated how it displayed data. We found that Cayenne had many limitations concerning how data can be displayed. For example, Cayenne lacked a map feature for displaying the geographic location of different sensors. After a lot of research, we came across Tago, which had a map feature and allowed for many different methods of data display.

Another consideration was the user experience in installing, using and viewing our product. We viewed this project to be expandable in the future. Furthermore, many sensors could be installed within UBC, each having the ability to measure a building's water quality. Thus, the organization of data was essential. In Tago, we initially hoped to have a map that displayed the water quality at each location, but we have found that Tago's map did not allow this functionality. It could only display the location of each sensor, which was still very useful.

Thus, our next goal was to design a dashboard for each sensor with its current data, the graph of previous data, and location data. Tago allowed the creation of these very easily with their widgets.

3.3 Results

On Tago, we created tabs allowing users to select which data category they would like to see. Figure 3.1 shows the image of the tabs. Figures 3.2 and 3.3 show the NTU and temperature gauge.

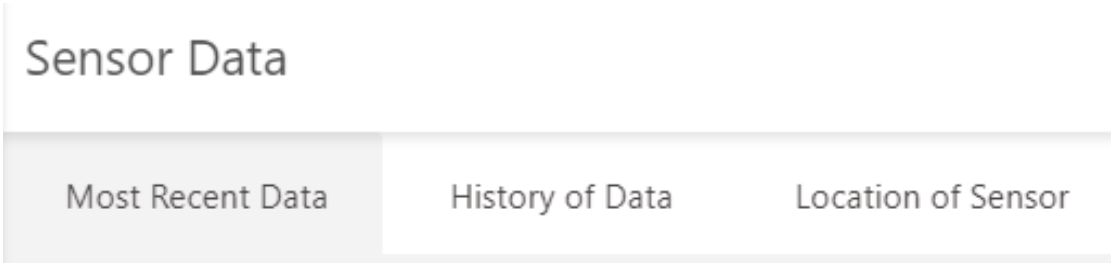

Figure 3.1 Tabs for Grouping Data

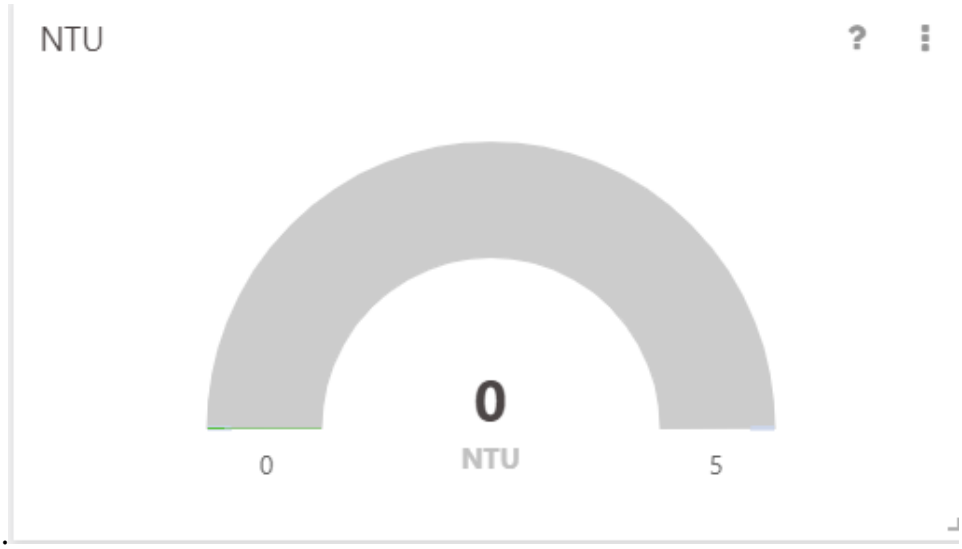

Figure 3.2 NTU Gauge for Most Recent Data

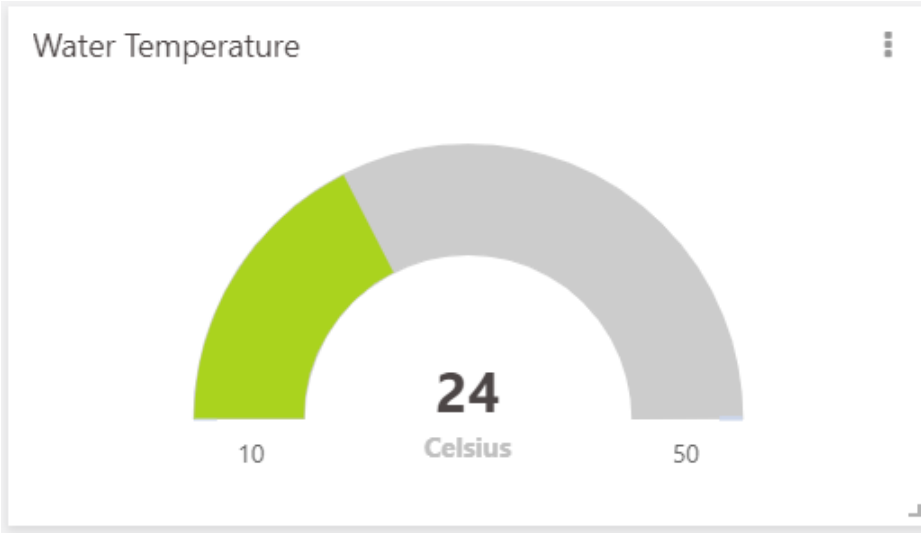

Figure 3.3 Water Temperature Gauge for Most Recent Data

The next tab showed the history of data measurements at a sensor. This is a compilation of figure 3.2 and 3.3, over the timestamp generated when the data were received. These graphs can be seen in Figure 3.4 and 3.5. Figure 3.6 showed the raw data in a table for troubleshooting.

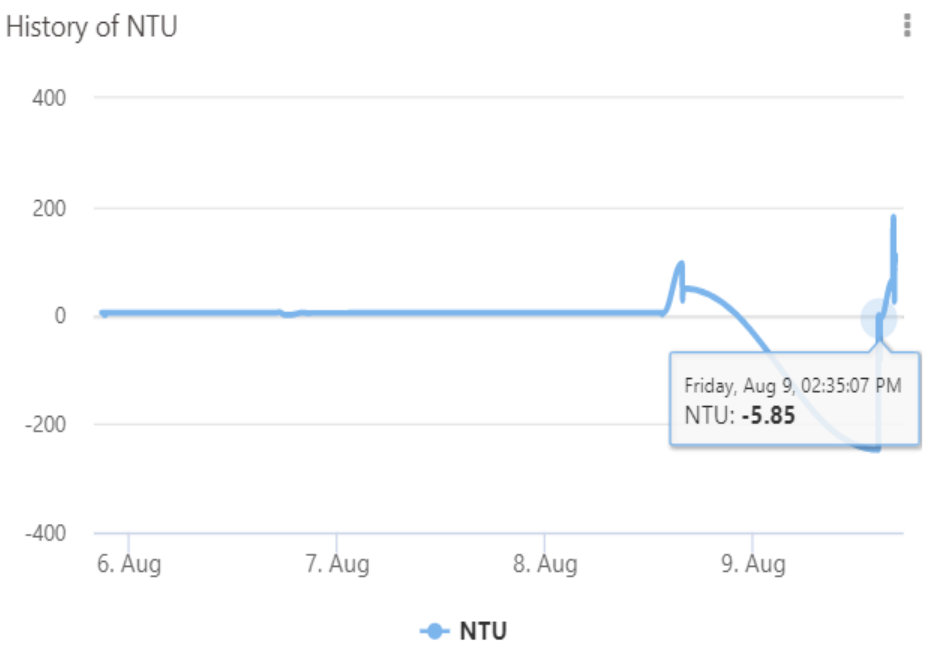

Figure 3.4 History of NTU Data on TagoIO

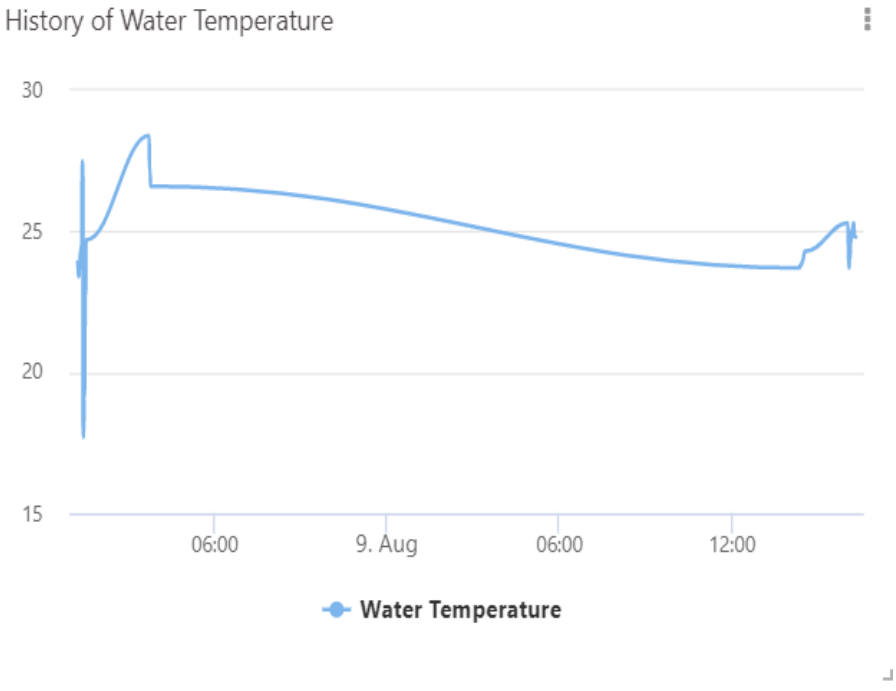

Figure 3.5 History of Water Temperature Data on TagoIO

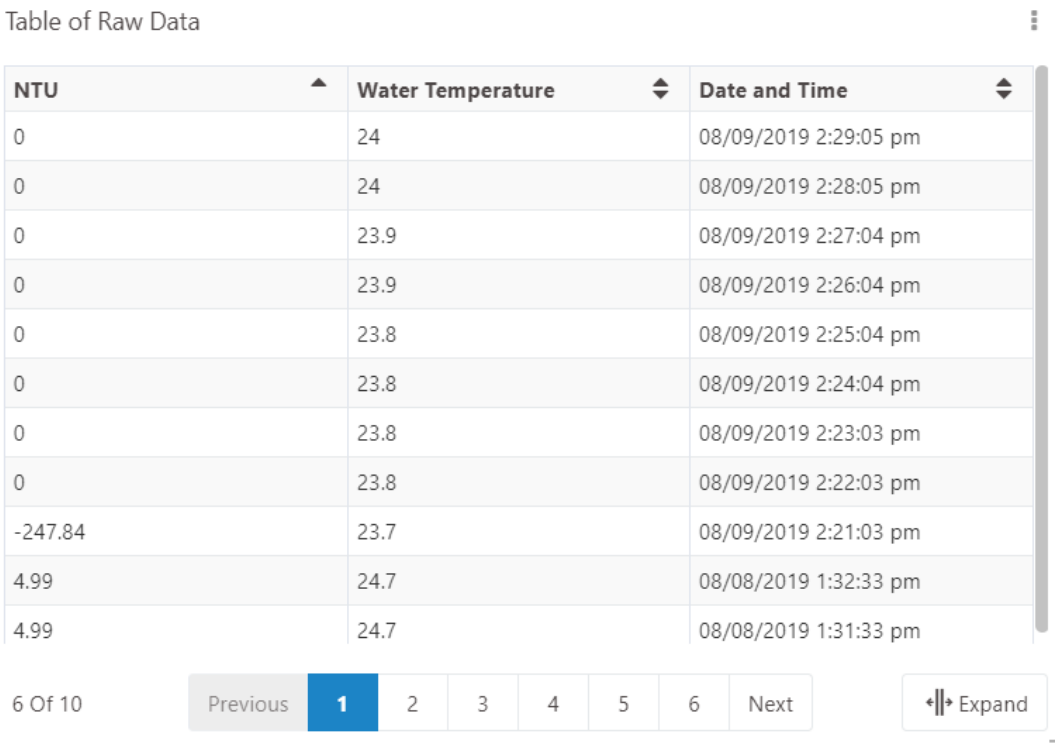

Figure 3.6 Table of Received Raw Data

Lastly, figure 3.7 shows the location of the sensor. This included a filter functionality to search the map by the names of the sensor.

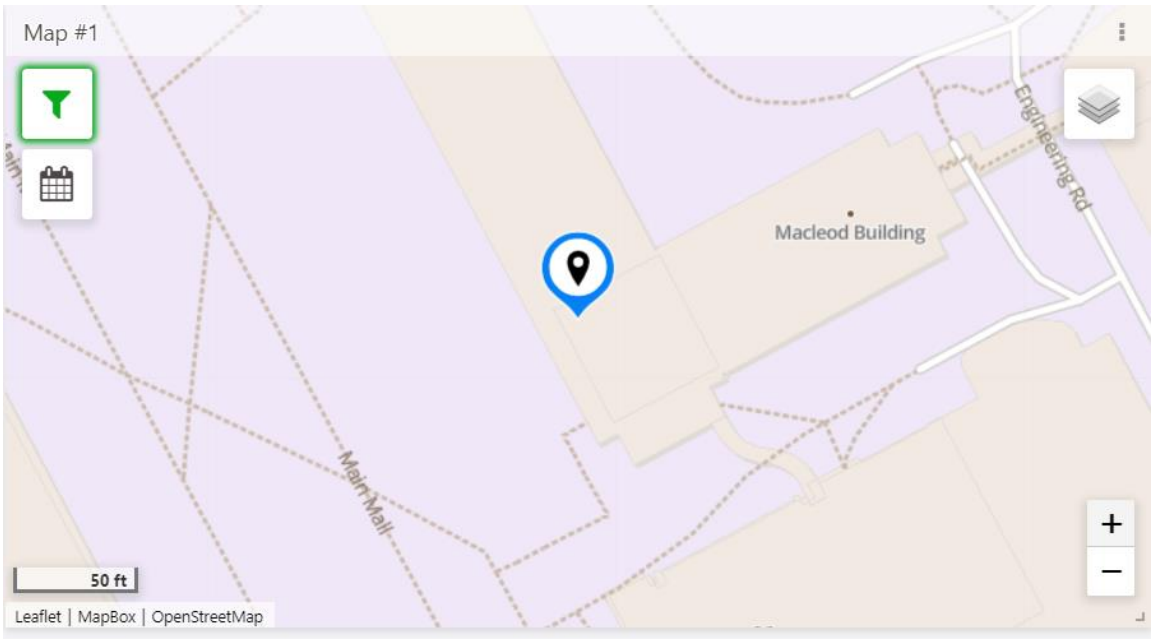

Figure 3.7 Map on TagoIO

3.4 Discussion

3.4.1 Turbidity and Temperature Visual Displays

For the visualization of the turbidity and temperature data, we wanted to focus on the standards for the safe drinking water quality. From research, we had found that WHO has defined a safe drinking water NTU to be near 1 and should not exceed 5 [5]. Similarly, the water temperature of aesthetic drinking water was defined to be 15 degree Celsius by the Guidelines for Canadian Drinking Water Quality [2,3]. Thus, for the tab named “Most Recent Data”, we had put two gauges representing the latest received data on TTN. The NTU gage had a range between 0 and 5, which had a green color between 0 and 1.5 NTU, yellowish-green between 1.5 and 4, and light red until 5 NTU, and dark red for higher NTU values. These colors transiently changed as the NTU values changed indicating that NTU up to 1 is safe = green, NTU > 5 is a bad = red drinking water quality. Similarly, the water temperature had a range between 10 and 50-degree Celsius. This indicates a safe drinking water temperature between 10 degrees and 15-

degree Celsius, while anything higher posed risks. The image of the gauge can be seen in figure 3.2 and figure 3.3

3.4.2 Sensor Map

The sensor map only showed the location of one sensor in Macleod Building at UBC because we only had one Arduino board and a pair of sensors available. If there were more sensors available, they would have been shown on the map. They could have been filtered according to their individual sensor name. However, the location of each sensor would have to be manually entered as we lacked the budget to buy GPS modules. Thus, hard-coding the latitude, longitude, and altitude were required for our current implementation as seen in Appendix A.

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

The results of this project will be briefly summarized and discussed in this section.

First, our project hardware, which was made up of the turbidity and temperature sensors, an LCD display, button, and the Arduino board allowed for a convenient way of accomplishing our design goals. However, a driver board for the turbidity sensor would have made the data retrieval process much easier. The presence of an LCD display and button increased the user-friendliness of our project. 3D-printed cases also improved the presentation of our project. Lastly, LoRa communication problems seemed to stem from the UBC gateway and not our antenna or transmitter module.

Second, the Arduino software that was written to make our project functional worked very well. The code that was written to transmit data from the sensors to TTN was readable and well-organized. The state machine format that we engineered to manage our project's control-flow also allowed for easier debugging.

Lastly, the Tago front-end web application that we employed provided better visualization of data than applications like Cayenne. Turbidity and temperature sensor data were shown with a gauge, graph, and a table. These allowed for monitoring of most recent data, historical data, and raw data. Regarding our map feature, it would have been much more user-friendly if water quality information could be displayed on the map as opposed to it just

displaying the location of each sensor. Data from a GPS module would have also been convenient as we would not have to keep hard-coding the location of each sensor for the map to display.

Recommendations

One major recommendation is to ensure that the UBC gateway is working properly before beginning the project. Our group wasted a lot of time trying to connect to TTN and as a result, troubleshooting our software became very tedious and inefficient.

Another recommendation is to ensure that enough time and budget is allocated for this project so proper sensors can be ordered and delivered on time. Students would also not have to worry about their financial status. With enough time and budget, we could have incorporated a pH sensor, dissolved oxygen sensor, and GPS sensor into our project. Thus, we could have built a final project with more complexity and creativity.

REFERENCES

[1] Christopher D. Kelley, Alexander Krolick, Logan Brunner, Alison Burklund, Daniel Kahn, William P. Ball, Monroe Weber-Shirk, "An Affordable Open-Source Turbidimeter", 2014, Retrieved from: https://www.mdpi.com/1424-8220/14/4/7142/pdf

[2] Government of Canada. (2009, Feb. 6). Guidelines for Canadian Drinking Water Quality: Guideline Technical Document - Temperature. Retrieved from https://www.canada.ca/en/health-canada/services/publications/healthy-living/guidelines-canadian-drinking-water-quality-guideline-technical-document-temperature.html?wbdisable=true

[3] Health Canada. (May 1979, Reprinted 1995). Temperature. Retrieved from https://www.canada.ca/content/dam/canada/health-canada/migration/healthy-canadians/publications/healthy-living-vie-saine/water-temperature-eau/alt/water-temperature-eau-eng.pdf

[4] J. Tölgyessy (11 March 1993). Chemistry and Biology of Water, Air and Soil: Environmental Aspects. Elsevier. pp. 297–. ISBN 978-0-08-087512-5. Retrieved 5 May 2013.

[5] LennTech, "Turbidity", 2019, Retrieved from: https://www.lenntech.com/turbidity.htm#ixzz3R3yPreK7

[6] Miroslaw Jonasz and Georges R. Fournier, *LIGHT SCATTERING BY PARTICLES IN WATER*, Academic Press, 2007, Retrieved from: https://www.sciencedirect.com/book/9780123887511/light-scattering-by-particles-in-water

APPENDIX A: ARDUINO SOFTWARE CODE

This appendix provides the Arduino code that was uploaded to our MKRWAN 1300 board. The code implements state machine to measure, average, and send the sensor data to TTN using lora.

// Some portion of this code is Gonazalo Casas's codes of mkrwan_02_hellow_world.ino under MIT License (specifically lines in init_lora() and connect_TTNs()). Copyright (c) 2018 Gonzalo Casas

// Rest are written and copyrighted by Young Hoon Ji (36400166), Craig Bonamis (20474145), Ammar Rehan (10649151), Tonny Li (19313048) of Group 6 ELEC391 in 2019 Summer 2 Semester of UBC Electrical Engineering. The use, copy, modify, merge, publish, distribute, sublicense, and/or sell of this code is not permitted without consent of the owners (Young Hoon Ji, Craig Bonamis, Ammar Rehan, and Tonny Li).

```
#include <MKRWAN.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <CayenneLPP.h>
#include <LiquidCrystal.h>
// ----- DEFINES ----- //
// Data wire is plugged into pin 2 on the Arduino
#define ONE_WIRE_BUS A2
#define Turbidity_in A3
```

```
// Button Location
#define BUTTON 6
// GPS LOCATION
#define LATITUDE 49.261694
#define LONGITUDE -123.249524
#define ALTITUDE 0
// ---------------------------- //
// ----- LORA INITIAL ----- //
// Select your region (AS923, AU915, EU868, KR920, IN865, US915, US915_HYBRID)
_lora_band region = US915;
// Copy these keys from The Things Network console and paste them here:
const char *appEui = "70B3D57ED00202DA";
const char *appKey = "8D0BD2BAB7AC43B8335635999703738B";
// ------------------------------------------ //
// ----- Global Variables ----- //
// Lora Globals
float temperature;
float NTU;
float avg_ntu = 0.0;
float avg_temp = 0.0;
float sum_ntu = 0.0;
float sum_temp = 0.0;
float latitude = (float) LATITUDE;
```

```
float longitude = (float) LONGITUDE;
float altitude = (float) ALTITUDE;


// Loop Globals
int loop_counter = 0;
int data_counter = 0;
int connect_loop = 1;
static int state;
bool b_flag = LOW;
bool start_stop_flag = LOW;
// ------------------------------------ //
// -------------------- //
// Current Payload Format uses CayenneLPP
CayenneLPP lpp(51);
LoRaModem modem(Serial1);
// Setup a oneWire instance to communicate with any OneWire devices
// (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);
 // Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(5, 4, 3, 2, 1, 0);
// ------------------------------------ //
```

```
// --------- SETUP CODE ------- //
void setup()
{
  Serial.begin(115200);
  // SET-UP BUTTON and LCD
  setup_button();
  setup_lcd();
  state = 0;
}
// ------ LOOP CODE ----- //
void loop()
{
  switch (state)
  {
    case 0: // Wait Screen
    {
      // LCD "Push Button to Uplink" Screen
      b_flag = LOW;
      Serial.print("State is:");
      Serial.println(state);
      wait_for_uplink_msg();
      delay(1000);
      break;
```

```
}
case 1:
{
  Serial.print("State is:");
  Serial.println(state);
  // Connect, check for abort
  start_uplink();
  break;
}
case 2:
{
  Serial.print("State is:");
  Serial.println(state);
  // MAIN LOOP FOR 30 SECONDS
  measure_temp_volt();  // Every 30sec, measure Current Data, add it to sum
  send_data_1_min();    // Every 1 minute, Sending avg data to TTN
  avg_data_2_min();     // Every 2 mins, we average sum, update the avg variables
  display_data_msg(temperature, NTU);
  delay(30*1000);
  break;
}
default:
{
```

```
      Serial.print("State is:");
      Serial.println(state);
      Serial.println("FATAL State Machine ERROR");
      state = 0;
      delay(10*1000);
      break;
    }
  }
}
// -------------- Messages ------------ //
void abort_msg()
{
  Serial.println("Process Aborted");
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("ABORT!!");
  lcd.setCursor(0, 1);
  lcd.print("Please wait 1min");
}
void wait_for_uplink_msg()
{
  Serial.println("Push Button to Begin Uplink");
  lcd.clear();
```

```
  lcd.setCursor(0, 0);
  lcd.print("Push Button");
  lcd.setCursor(0, 1);
  lcd.print("To Begin Uplink");
}
void connecting_to_TTN_msg()
{
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Connecting");
  lcd.setCursor(0, 1);
  lcd.print("to TTN...");
}
void success_TTN_msg()
{
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Connection");
  lcd.setCursor(0, 1);
  lcd.print("Success!!");
}
void display_data_msg (float temperature, float NTU)
{
```

```
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Temp.: ");
  lcd.print(temperature,2);
  lcd.setCursor(0, 1);
  lcd.print("NTU:   ");
  lcd.print(NTU,2);
}
void display_sending_msg (int process)
{
  lcd.clear();
   if (process == 0)
  {
    lcd.setCursor(0, 0);
    lcd.print("Sending Message");
    lcd.setCursor(0, 1);
    lcd.print("to TTN");
  } else if (process == 1)
  {
    lcd.setCursor(0, 0);
    lcd.print("Message Sent");
    lcd.setCursor(0, 1);
    lcd.print("Success!!!");
```

```
  } else if (process == 2)
  {
    lcd.setCursor(0, 0);
    lcd.print("Failed...");
  }
}

void reconnecting_in_1_min_msg ()
{
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Failed:Reconnect");
  lcd.setCursor(0, 1);
  lcd.print("In 1 Minute    ");
}
// -------------------- Functions ----------------- //
void start_uplink()
{
   // Init Sensor
   init_sensors();
   // Connect to TTN
   connect_TTN();
}
```

```
void setup_button ()
{
  // Setup Button
  Serial.println("Setting up Button");
  pinMode(BUTTON, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON), ButtonHandler, CHANGE);
}
void ButtonHandler ()
{
  while (digitalRead(BUTTON) == LOW)
  {
    delay(50);
  }
  Serial.println("------ Button Pressed!! ------");

  switch (state)
  {
    case 0:
    {
      if (b_flag == LOW) state = state + 1;
      break;
    }
    case 1:
```

```
    {
      state = 0;
      abort_msg();
      b_flag = HIGH;
      break;
    }
    case 2:
    {
      state = 0;
      abort_msg();
      b_flag = HIGH;
      break;
    }
    default:
    {
      state = 0;
      break;
    }
     Serial.println("------ ENd of button Handler! ------");
  }
}
void setup_lcd()
{
```

```
  Serial.println("Setting up LCD");
  lcd.begin(16, 2);
  // Print a message to the LCD
  lcd.setCursor(0, 0);
  lcd.print("Welcome!");
  delay(4000);
}
void init_lora ()
{
  Serial.println("Initializing LORA");
  while (!Serial);
  if (!modem.begin(region)) {
    Serial.println("Failed to start module");
    while (1) {}
  };
  Serial.print("Your device EUI is: ");
  Serial.println(modem.deviceEUI());
}
void connect_TTN ()
{
  // Connection Loop
  while (connect_loop && (state == 1))
  {
```

```
 init_lora();
 connecting_to_TTN_msg();
 Serial.print("Connecting... \n");
 int connected = modem.joinOTAA(appEui, appKey);
 Serial.print("State while Connecting is:");
 Serial.println(state);
if (state == 0) return;
if (!connected) {
  Serial.println("Something went wrong; are you indoor? Move near a window and
retry");
  Serial.println("Reconnecting in 1 minute");
  reconnecting_in_1_min_msg();
  delay(60*1000);
  modem.restart();
 }
 else
 {
  Serial.println("Successfully joined the network!");
  success_TTN_msg();
  connect_loop = 0;
  state = state + 1;
 }
}
```

```
 if (state == 1)
  {
    Serial.println("Enabling ADR and setting low spreading factor");
    modem.setADR(true);
    modem.dataRate(5);
  }
}
void init_sensors()
{
  // Get Initial Temperature
  Serial.print("-- Requesting temperatures... \n");
  sensors.requestTemperatures();      // Send the command to get temperatures
  Serial.print("-- Initial Temperature at index 0 is: ");
  avg_temp = sensors.getTempCByIndex(0); // Why "byIndex"?
  Serial.print(avg_temp);
  // Get current Turbidity Voltage
  avg_ntu = (float) analogRead(Turbidity_in);
  avg_ntu = -20.158 * avg_ntu + 17638.25;
  if (avg_ntu < 0.0) avg_ntu = 0.0;
  if (avg_ntu > 5000.0) avg_ntu = 5000.0;
  Serial.println("\n-- Initial Turbidity is: ");
  Serial.println(avg_ntu);
  Serial.println("\n----------------------------------------");
```

```
}
void measure_temp_volt ()
{
  // Get Temperature
  Serial.println("\n-- Requesting temperatures...");
  sensors.requestTemperatures();      // Send the command to get temperatures
  temperature = sensors.getTempCByIndex(0);
  // Get current Turbidity Voltage
  NTU = (float) analogRead(Turbidity_in);
  Serial.print("ANALOG INPUT IS:");
  Serial.println(NTU);
  NTU = -20.158 * NTU + 17638.25;
  if (NTU < 0.0) NTU = 0.0;
  if (NTU > 5000.0) NTU = 5000.0;
  // Add to avg_temp, global variable
  sum_temp = sum_temp + temperature;
  // Add to avg_ntu, global variable
  sum_ntu = sum_ntu + NTU;
  Serial.print("\n----------------------------------------");
  Serial.print("\n-- Current Temp: ");
  Serial.print(temperature);
  Serial.print("\n-- Current NTU is: ");
  Serial.print(NTU);
```

```
  // LCD Printing
  display_data_msg(temperature, NTU);
  // Printing
  Serial.print("\n-- Sum Temp: ");
  Serial.print(sum_temp);
  Serial.print("\n-- Sum NTU: ");
  Serial.print(sum_ntu);
  Serial.print("\n----------------------------------------");
}
void send_data_1_min ()
{
  // Send the data every minute
  if (data_counter > 0)
  {
    // Update LPP Object
    lpp.reset();
    lpp.addTemperature(1, avg_temp);
    lpp.addAnalogOutput(2, avg_ntu);
    // GPS
    lpp.addGPS(3, LATITUDE, LONGITUDE, ALTITUDE);
    // Send it off
    Serial.print("\n----------------------------------------");
    Serial.print("\nSending Data");
```

```
Serial.print("\n-- Avg Temp: ");
Serial.print(avg_temp);
Serial.print("\n-- Avg NTU: ");
Serial.print(avg_ntu);
Serial.print("\n-- Longitude: ");
Serial.print(longitude);
Serial.print("\n-- Latitude: ");
Serial.print(latitude);
Serial.print("\n-- Altitude: ");
Serial.print(altitude);
Serial.print("\n----------------------------------------");
display_sending_msg(0);
modem.beginPacket();
modem.write(lpp.getBuffer(), lpp.getSize());
// Check Error
int err = modem.endPacket(false);
if (err > 0)
{
  Serial.println("\n!Big success!");
  display_sending_msg(1);
} else
{
  Serial.println("\nERROR: END PACKET RETURNED FALSE");
```

```
      display_sending_msg(2);
    }
    // Reset Data_Counter
    data_counter = 0;
  } else
  {
    data_counter++;
  }
}
void avg_data_2_min ()
{
  // Retrieve data for 4 * 30 seconds = 2 minutes
  if (loop_counter > 2)
  {
    // Take Average (divide by 4)
    avg_temp = sum_temp / 4.0;
    avg_ntu = sum_ntu / 4.0;
    // Reset Loop Counter, sum_temp, sum_ntu
    loop_counter = 0;
    sum_temp = 0;
    sum_ntu =0;
  } else
  {
```

```
    // Add to Loop Counter
    loop_counter++;
  }
  Serial.print("\n-- Data Counter is: ");
  Serial.print(data_counter);
  Serial.print("\n-- Loop Counter is: ");
  Serial.print(loop_counter);
  Serial.print("\n----------------------------------------");
  Serial.print("\n----------------------------------------");
}
```