IOT-ENABLED SMART SENSING SYSTEM

Nawshin Nazrul

Bachelor of Engineering
Computer Engineering Major

Department of Computer Engineering
Macquarie University

November 02, 2017

Supervisor: Professor Dr. Subhas Chandra Mukhopadhyay
Co-supervisor: PhD student Md. Eshrat E. Alahi

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my project supervisor, Professor Dr. Subhas Chandra Mukhopadhyay for his guidance and support throughout the thesis project. I would also like to extend my respects to my co-supervisor, PhD student Md. Eshrat E. Alahi for his kind help and cooperation in every step from the beginning to the end of this project. Finally, I would like to thank my wonderful parents who have taught me to love and pursue my passion till the end without fearing the obstacles in the way in-between.

STATEMENT OF CANDIDATE

I, (Nawshin Nazrul), declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Computer Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment an any academic institution.

Student's Name: Nawshin Nazrul

Student's Signature: NAWSHIN NAZRUL

Date: 02 November 2017

ABSTRACT

Nowadays, various types of smart sensing systems are available worldwide due to the remarkable advancement achieved in the field of sensor technology. Not many of them use capacitive sensors that need complex circuitry as well as software algorithm. This project aims to develop an unconventional state of the art smart sensing system for environment monitoring. The system will incorporate a complex sensor node compatible with different kinds of capacitive sensors besides resistive sensors that are not capable of communicating directly with the microprocessor. This will also be an IoT (Internet of Things) integrated system enabling it to continuously send the sensor data to a designated web server for constant monitoring of the environment. The idea is to overcome the challenges of building a composite smart sensing system using complex algorithm. For this particular project, three different kinds of sensors (LM335Z, HS1101LF and TGS2600) have been used in order to sense various attributes of our surrounding environment. LM335Z and TGS2600 are simple resistive types of sensors that have the capabilities to measure temperature and general air quality respectively. On the other hand, HS1101LF is a capacitive type of sensor that can measure humidity and temperature. As mentioned earlier, interfacing capacitive sensors to the microcontroller is not as simple as interfacing resistive ones. Therefore, this project focuses on developing sensor nodes incorporating both resistive and capacitive sensors in one connected circuit and achieve correct sensor data that will be later uploaded to an IoT server (ThingSpeak Channel) for continuous monitoring of the environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart sensors have become an inevitable part of our society by providing security and safety with constant monitoring of our health and environment [8]. Previous studies show that, smart sensors possess the capability of completely changing the method used to maintain, control and monitor the civil infrastructure [9]. However, developing a smart sensing system including low-power, low-cost as well as small sized sensors and which supports wireless communication is still a great challenge for the researchers. Therefore, the goal of this project is to build a low-power and cost-effective system including IoT applications for environment monitoring purpose. Currently, most of the available smart sensing systems are quite straightforward and only include resistive type sensors. This project focuses on establishing a compound sensing system incorporating both capacitive and resistive sensor in one circuit. It will also be connected to a web-server where the processed sensor data will be sent for constant monitoring.

The sensor node developed for this project is a complex node including two resistive sensors for measuring temperature and general air quality (LM335 and TGS2600) as well as one capacitive sensor (HS1101LF) for measuring humidity of the surrounding environment. An impedance analyser (AD5933) is also introduced in the project to create communication between humidity sensor and the main Arduino microcontroller via I^2C protocol. A detailed description of the system components and methodology to achieve accurate results have been discussed in the following chapters. The complete circuit diagram has also been provided along with the figure of the working sensor node. After processing the sensor data, they are sent straight to the IoT server for monitoring purpose. For this, an open-source IoT platform called "ThingSpeak" has been selected since creating a private channel here and uploading data constantly are more straightforward than other IoT applications. Finally, the field experiment results and other challenges encountered during the project are illustrated at the end of this document.

1.1 Overview of the Project

This section outlines a brief discussion of the overall project and the contents of this document. This project is supervised by Professor Dr. Subhas Chandra Mukhopadhyay and co-supervised by PhD student Md. Eshrat E. Alahi. The overall thesis project is divided into two units ENGG460 and ENGG411. Each of these units include 13 weeks. During the first 4 weeks of ENGG460, the list of projects was made available to the students and the students were expected to choose their project, meet the prospective supervisor and submit their list of preferred projects online. The list of project allocations was available by the end of week 6. After that, a regular meeting per week with the supervisor has been setup to discuss any problem encountered during working on the project in order to ensure that the progress is keeping up with the schedule. Before the commencement of ENGG411, there was a winter vacation of four weeks and a few milestones were achieved during the vacation. Finally, ENGG411 started from July 31st, 2017 and the final report submission deadline is on 6th November 2017. The framework of the thesis project is as follows:

Chapter 2 illustrates some important definitions, background and literature review related to this project. Definitions of smart sensor and different types of sensors are provided and the literature review section contains information collected from the research conducted during the last few weeks of ENGG460.

Chapter 3 discusses the proposed design of this project. It includes the hardware and software architecture of the whole system.

Chapter 4 depicts the experimental setup and working procedures for this project. It explains the whole procedure step by step following which the expected results were attained.

Chapter 5 includes the results and discussions which illustrates the results achieved during the field experiment as well as refer to the problems encountered during the process and the solutions that could be thought of up till now.

Chapter 6 will conclude the project by outlining the goals achieved and the summary of the project. It will also discuss any future work that can take place after what has been done till now to achieve further improvements.

Project Objectives:

- Getting familiarised with Arduino Uno Wi-Fi platform and Arduino IDE

- Gain experience in embedded C, Matlab programming and netwroking
- Understanding characteristics of different types of sensors
- Developing a low-cost and low-power smart sensing system connected to an IoT server for constant monitoring of environment

1.2 Time and Financial Budget Overview

The project started from 24th of April, 2017 which was the beginning of the 7th week of ENGG460. From then the number of total weeks till now is 22, excluding 2 weeks of exam period. The project plan or the project timeline is provided in Appendix A. The project timeline planned during the first week had been followed closely and more than 50 percent of work had been accomplished within week 6. Therefore, the project has been completed successfully by due date.

Every student is allocated with approximately 400 Australian dollars as the financial budget for the thesis project. Some of the equipment were already provided in the laboratory, for example, Arduino Uno Wi-Fi, resistors, jumper wires, USB cables, soldering irons etc. However, most of the important equipment needed to be bought. The table below shows the items that were bought along with their price in Australian dollar -

Name of The Equipment	Price (Per Item in Australian Dollar)
Impedance Component Analyser (2)	$50.0*2 = 100.0$
Temperature Sensor (LM335) (2)	$1.49*2 = 2.98$
Humidity Sensor (HS1101LF) (2)	$8.85*2 = 17.7$
Gas Sensor(TGS2600) (2)	$25.0*2 = 50.0$
Multiplexer (ADG849) (2)	$2.89*2 = 5.78$
Power Bank (Cygnett 4,000 mAh) (2)	$39.95*2 = 79.9$
Total	=256.36

Table 1.1: Price of the items needed to be purchased

Chapter 2

Background and Literature Review

This chapter contains some important definitions, background theory and literature review related to this project. It describes what a smart sensor is and what are the main kinds of a smart sensor. Later, it explains the working principles of resistive and capacitive sensor. Finally, it discusses the advantages and drawbacks (if any) of some of the state of the art smart sensing systems available in the market.

2.1 What is a Smart Sensor?

Ideally, a smart sensor indicates to a sensing element with enhanced abilities which is usually provided by a microprocessor. In other words, Smart sensors are regular sensing elements with embedded intelligence that can transmit processed information to external users [8]. These sensing elements can be resistive, capacitive, piezo resistive materials or even photodiode. Smart sensors not only are capable of self-health assessment or self-calibration but also provide information with increased integrity and reliability. The main difference between a smart and an ordinary sensor is that the smart sensor has a microprocessor instead of having a sensor interface, which can perform various intellectual activities, such as analog to digital conversions, signal conditioning, interfacing and making other decisions. The figure below refers to the block diagram of a smart sensor [9]-

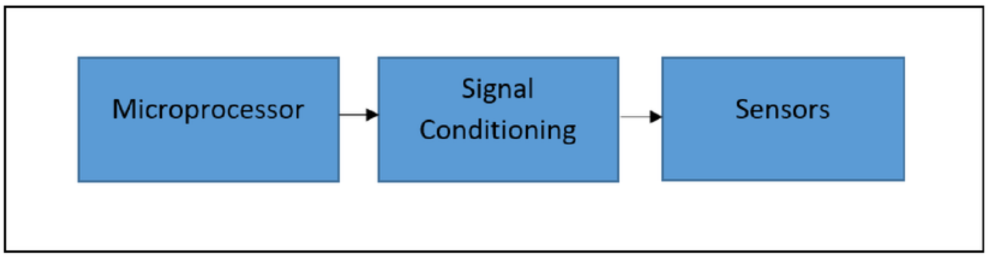


Figure 2.1: Block diagram of a Smart Sensor

2.2 Types of Sensors

There are different kinds of sensors based on various requirements of the users, like temperature sensor, humidity sensor, pressure sensor, gas sensor, light sensor and so on. However, all these sensors can again be divided into several types depending on their applications, type of input voltage and corresponding circuitry. This project mainly focuses on resistive and capacitive types of sensors since they are the most common types to be used in smart sensor systems. The functional attributes of these two sensors are explained below-

Resistive Sensors [10]:

Resistive sensors are most frequently used in smart sensing systems due to its user-friendly functionality. These perform measurements by detecting the change of resistance in a resistive element. The following equation measures resistance:

$$R = \frac{\rho \times L}{A} \tag{2.1}$$

Where,

R = resistance of the material;
ρ = electrical resistivity of the material;
L = cross-sectional length of the material;
A = cross-sectional area of the material.

Capacitive Sensors [10]:

Capacitive sensors are well-known for their ability to store energy. It basically follows the operating principle of coupling effect. According to this principle, dielectric materials are placed between two poles that contain opposite charges and the permittivity of dielectric materials influences the intensity that polarises the charges of these materials. The capacitance here depends both on the sensors size and the permittivity of the material used. The equation for that is given below:

$$C = \frac{\epsilon_0 \times \epsilon_1 \times A}{d} \tag{2.2}$$

Where,

C = capacitance of the interdigital sensor;
ϵ_0 = the permittivity of vacuum;
ϵ_1 = relative permittivity;
A = the effective area;
d = the effective spacing between electrodes of different polarity.

The main operational difference between resistive and capacitive sensor is the type of input voltage. Resistive sensors can operate at DC voltage whereas capacitive sensors can only operate at AC supply voltage as an input.

2.3 Literature Review

As mentioned earlier, the applications of smart sensor systems are becoming more and more popular each day. Specially, with the remarkable advancement made in the past few years in the field of wireless sensor networks (WSN) has made it even more challenging to develop sensor nodes with more advanced and newer features and attributes. This section discusses some of the state of the art smart sensing systems, their special characteristics and drawbacks.

A smart nitrate sensor system has been developed in [11] which is also an IoT-based system. This system used interdigital capacitive sensors for measuring nitrate concentration in surface and ground water. Arduino Uno Wi-Fi was introduced as the sensor platform that contains Atmel ATmega328 microprocessor. The most remarkable features of this design were that it could successfully measure nitrate concentration ranging from 0.01-0.5 mg/L and the sensor data was sent to an open database platform called ThingSpeak using HTTP POST protocol to make it a connected system. However, this sensor system could only be used for low measurements.

Another IoT-enabled smart sensor design has been proposed in [12] for domestic environment monitoring. This system contained three different sensing components for measuring various parameters of home environment with almost 97 percent accuracy of reliability of the provided information. A resistive temperature TMP 36 and a light sensor BPW 21R are some of the sensors that were used in this system. For IoT configuration, it used ZigBee Wireless Sensor Network and XBee-S2 module as the gateway. Although it achieved a remarkable amount of reliability, occasional changes were noticed during the throughput process because of interferences from other networks.

A barometric sensor and an alcohol sensor (BMP 085, MQ 3) were used in designing a detachable smart sensor in [13] for environment monitoring. Instead of Arduino, it used DSTIM (Detachable Smart Transducer Interface Module) sensor platform and PIC 18LF2550 as the microcontroller. As the previous design, it also used ZigBee Wireless Sensor Network for enabling IoT configuration. One of the noteworthy features of this system is it can be distributed at different geographical locations and measure as well as send information about the environment to the server. The only weakness of this system is it has a comparatively low power and sensing efficiency.

[14] Describes the design of a wireless sensor node for Biomedical Research which used SHIMMER as the sensor platform and TI MSP430 as the microcontroller. This is a very

flexible platform that has the capability of physiological, kinematic and ambient sensing. One of the problems with this system is it uses Bluetooth connection which is often not very reliable. Also, it is still an ongoing research.

A low-cost wireless sensor system is developed using Arduino and Raspberry Pi platform [15]. It consists of a base station and a number of sensor nodes that can be distributed in different locations. The base station uses a low-power and small Raspberry Pi Model B based on ARM processor. On the other hand, the sensor nodes contain Arduino Uno R3 platform integrated with Atmel Atmega328 microcontroller. Wireless communication is enabled among the sensor nodes and the base station via XBee Pro S2B module which will send the data to a MySQL databse. However, the storage capacity in this system is not large enough. RHT 03 (temperature and humidity sensor) is used here to measure humidity and temperature of the environment.

Chapter 3

System Description

As mentioned earlier in the document, the goal of this project is to develop a couple of sensor nodes consisting of LM335 temperature sensor, TGS2600 general air quality sensor and HS1101LF humidity sensor as the main sensors and Arduino Uno Wi-Fi as the main sensor platform which is based on ATmega328P microcontroller. LM335 and TGS2600 being resistive sensors, interfacing them to the microcontroller is quite straightforward since resistive sensors can function at DC input voltage provided by Arduino microcontroller. However, capacitive resistors only support AC input voltage which cannot be provided by Arduino microcontrollers. Therefore, an impedance analyser (AD5933) has been used in this project to solve this problem which makes the whole system a little bit more complex. The humidity sensor is interfaced to the microcontroller via I^{2}C protocol which takes place in the impedance analyser. All necessary measurements are processed here and then sent to the microcontroller through this protocol. The detailed discussion about this protocol will be given later in this document. After processing all the sensed data through software algorithm, the information is uploaded in a designated web server through IoT configuration. Arduino Uno Wi-Fi has its Wi-Fi module (ESP8266) which has an integrated TCP/IP protocol stack that helps to get access to the Wi-Fi. Dragino (LG-01 S) has been used as the IoT gateway that will connect the Arduino Uno Wi-Fi with the ThingSpeak server which is an open database platform. The processed data will be sent directly to a ThingSpeak channel constantly where the user can monitor the updated information.

A block diagram of the design is given in the next page.

3.1 Sensor Node

As mentioned above, the sensor node consists of the sensors (HS1101LF, LM335 and TGS2600), a multiplexer (ADG849), Arduino platform (Arduino Uno Wi-Fi) and the impedance analyser. To provide power, a low-cost power bank has been used in this project. A brief discussion about these components are outlined in the sections below-

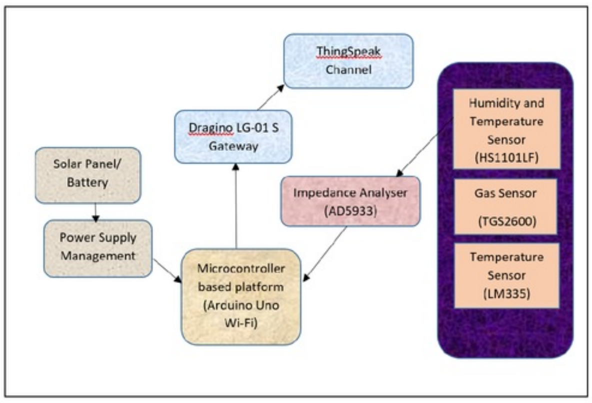

ThingSpeak Channel
Dragino LG-01 S Gateway
Solar Panel/ Battery
Power Supply Management
Microcontroller based platform (Arduino Uno Wi-Fi)
Impedance Analyser (AD5933)
Humidity and Temperature Sensor (HS1101LF)
Gas Sensor (TGS2600)
Temperature Sensor (LM335)

Figure 3.1: A simple block diagram of the proposed design

3.1.1 HS1101LF Module - Relative Humidity Sensor

This is an environment-friendly low-cost relative humidity sensor with high reliability and long-term stability. It has a really low temperature coefficient which also does not need any calibration in standard conditions. It has an operating temperature ranging from -60°C to 140°C. The supply voltage should be 10 VAC. Since it cannot operate at DC voltage and does not have I^2C bus or pins, the measurements are done using a separate impedance analyser which is then sent to the Arduino microcontroller via I^2C protocol. The impedance analyser first measures the unknown impedance of the sensor which is then used to calculate the capacitance (This will be discussed later in detail). All the measurements need to be done at 10 KHz/V as per the datasheet of this sensor. This capacitance is crucial to finally measure the humidity using a polynomial equation which is also provided in the datasheet [16].

Figure 3.2: HS1101LF Relative Humidity Sensor

3.1.2 LM335 Module - Temperature Sensor

LM335 is a precision IC (Integrated Circuit) temperature sensor which provides temperature in degree Kelvin. This is low-voltage IC which operates at 3.3V. It has three pins, pin1 is the adjustable pin which is only used when calibration is needed to get more precise temperature readings. Pin 2 and 3 are the output and ground pin respectively. This sensor gives an output of 10 mV per degree Kelvin. Pin 2 is connected to an analog pin of the Arduino board and a 2KΩ resistor is connected in series with pin 2 and 3.3 V of the Arduino board. Arduino board can measure the analog voltage with suitable code and provide output temperature in Kelvin, Celsius or Fahrenheit. This sensor has an operating temperature ranging from -40°C to 100°C [17].

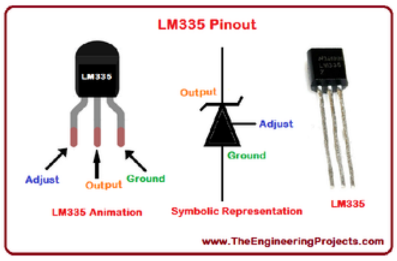


Figure 3.3: LM335 Temperature Sensor

3.1.3 TGS2600 Module - General Air Quality Sensor

TGS2600 is a low-power and low-cost general air quality sensor which is highly sensitive to gaseous air contaminants in low concentration, like hydrogen or carbon monoxide in cigarette smoke. The sensing element is made of a metal-oxide semiconductor layer generated on top of an aluminium substrate of the sensing chip along with an integrated heater. This heater helps maintaining the sensing unit at a pre-determined temperature to get precise output. The conductivity of the sensor increases in the presence of a detectable gas relying on the concentration of the gas in air. This change of conductivity can be converted to an output with necessary electrical circuit that can eventually provide a measure of that gas concentration. This is an analog sensor therefore an ADC (analog-to-digital converter) is needed to convert the data into expected output. This sensor has 4 pins; pin 1 is the VCC pin, pin 2, 3 and 4 are integrated heater, data and ground pin respectively [6].

Figure 3.4: TGS2600 General Air Quality Sensor [1]

3.1.4 Impedance Analyser

The impedance analyser used here is AD5933 which is a high precision impedance converter solution. The main components include an on-board 12-bit frequency generator, 1 MSPS and an analog-to-digital converter (ADC), a temperature sensor and I^2C interface. An external complex impedance gets excited at a known frequency by the frequency generator and the on-board ADC samples the received response signal from that impedance. Next the on-board DSP engine performs a discrete Fourier transform (DFT) on it and returns a real and imaginary value at each frequency increment. These values are stored in particular registers and can be read using I^2C protocol [2].

After calibration, a frequency sweep is performed and the magnitude as well as the phase shift are easily calculated at each frequency point using the following equation [2]

$$Magnitude = \sqrt(R^2 + I^2) \tag{3.1}$$

$$Phase = tan^{-1}(I/R) \tag{3.2}$$

Serial I^2C interface and I^2C Timing of AD5933

AD5933 is controlled via I^2C protocol and always acts as the slave device while connected through this bus interface with another device (master device). It has a 7-bit bus address which is by default 0001101 (0x0D) when it is powered up. According to the I^2C protocol, the data transfer is always initiated by the master device followed by a start condition. This start condition refers to the transition of the Serial Data Line (SDA) from high to low given that the Serial Clock Line remains high at that time. Then the slave device shifts in next 8 bits containing an R/W (0= write, 1= read) bit along with the bus address which decides the direction of data flow (if data is written to or read from the slave device).

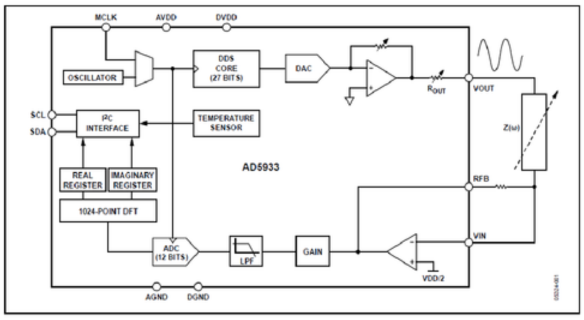


Figure 3.5: Functional block diagram of AD5933 impedance analyser [2]

Data transmission is always carried out in sequences of nine clock pulses, 8 bits of data which are followed by an acknowledge bit, either sent by the master or the slave device. Data transmission must occur during the low period of clock signal and remain stable when it is high since a low-to-high transition during the high period of clock signal might be acknowledged as a stop signal. The stop condition is established when all the data have been written to or read from the slave device [2].

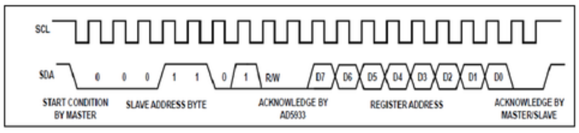


Figure 3.6: Timing diagram of AD5933 impedance analyser [2]

3.1.5 Microcontroller

Arduino Uno Wi-Fi has been chosen as the sensor platform for this project for the advantage of having its own integrated Wi-Fi module. It has ATmega328P microcontroller and ESP8266 Wi-Fi module integrated into it. Its operating voltage is 5 V and can be powered up by simply connecting it to a computer via USB cable. Arduino Uno Wi-Fi contains 6 analog input pins, 14 digital input and output pins, a USB connection, a reset button and so on for other functionalities. This board is popular for its small size, less weight and low power consumption [3].

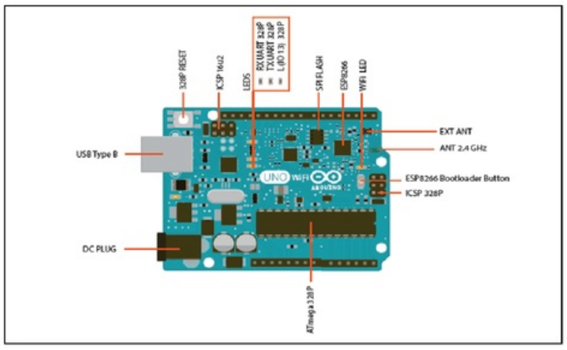


Figure 3.7: Arduino Uno Wi-Fi Board [3]

3.1.6 ADG849 Module - Switch

This is a rigid CMOS SPDT (single pole, double throw) analog switch that has an operating voltage ranging from 1.8 to 5.5 V. It has total 6 pins where pin 2 and 3 are VDD and ground pins respectively. Pin 1 is the logic control input which controls whether switch S1 (pin 4) or switch S2 (pin 6) should be on. This multiplexer is used in this project to switch between calibration and measurement when necessary for the humidity sensor [4].

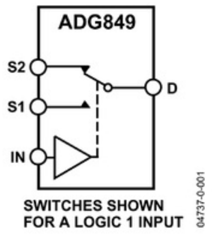


Figure 3.8: ADG849 Module [4]

3.1.7 Power Bank

For this project, Cygnett ChargeUP 4000 mAh power bank has been used as the main source of power. This has been designed following the latest Lithium polymer battery technology which has allowed its smaller cells to preserve more charge and provide higher output. Two or three cell phones can be charged at a time with this power bank. It is also quite small and slim (103g) which is suitable for the small sensor node. It has a digital display to show how much power it has at the moment. A micro-USB comes with the power bank which can be used to recharge it by plugging it into a computer, car charger or wall charger. The battery will also turn off automatically while not being used to save power. This power bank was able to provide power to the sensor node for 24 hours without having to recharge it [5].

Figure 3.9: Cygnett ChargeUp 4000 Power Bank [5]

3.2 Software Platform

The Arduino IDE (Integrated Development Environment) 1.8.4 has been used as the main software platform to upload necessary codes to the Arduino Uno Wi-Fi board. This software consists of a text editor for writing code, a text console, a message area, a toolbar with common functions as well as a wide range of libraries and examples helpful for a first-time user. It can be connected with any compatible Arduino hardware through a USB cable for uploading code and communicating with it. This is an open source platform that runs on Windows, Mac OS X and Linux. Its source code is developed by GitHub and the environment is written in Java and based on processing and other open-source software [18].

3.3 IoT Configuration

As it has been already mentioned earlier, Arduino Uno Wi-Fi has its own integrated Wi-Fi module (ESP8266) along with TCP/IP protocol stack which enables access to the internet. Since it covers a very small area, Dragino (LG-01 S) has been introduced as the IoT gateway to connect Arduinos Wi-Fi with a particular ThingSpeak channel so that the sensor data can be constantly updated there. REST protocol has been used in the code for transmitting data to the channel. ThingSpeak platform has been chosen for this project since it is free and very easy to create an account as well as a private channel here. The procedure of transferring data from client to the server is also quite straightforward. When a private channel is created, a unique API key and a channel number are automatically created for that particular channel. Arduino IDE has a built-in library for ThingSpeak and the code is uploaded to the Arduino board with the API key included in it. The channel can display as many fields as required and gets updated at a user defined interval.

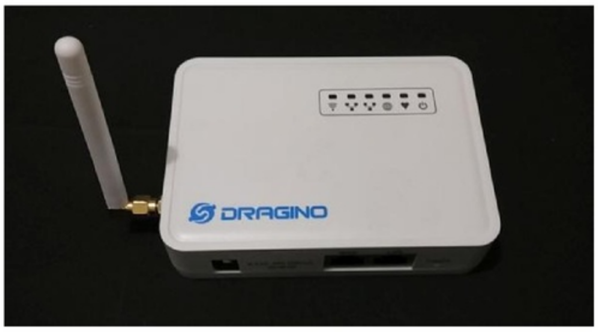


Figure 3.10: Dragino LG01-S Gateway

Chapter 4

Experimental Setup and Working Principles

This chapter discusses all necessary theories and methods required to get accurate results for the three sensors. At first, the calibration procedure, circuit diagram of the LM335 temperature sensor as well as the code to gain correct temperature results are described. After that, the working principles of HS1101LF humidity sensor is explained in detail. The calibration test of this sensor is a bit complicated. Once the calibration is performed, the unknown impedance of the sensor is calculated to get the capacitance of the sensor which is most important parameter to get accurate humidity results. Finally, the methodology to obtain TGS2600 air quality sensor is illustrated with necessary theories and graphs.

4.1 Circuit Diagram

The circuit diagram of the complete system as well as the picture of the sensor node is given in the next page:

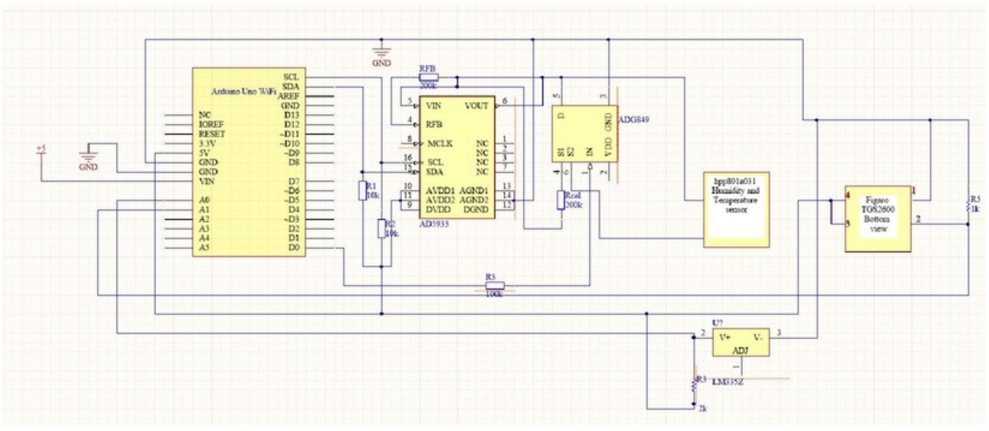


Figure 4.1: Circuit Diagram of AD5933 impedance analyser with Arduino Uno Wi-Fi

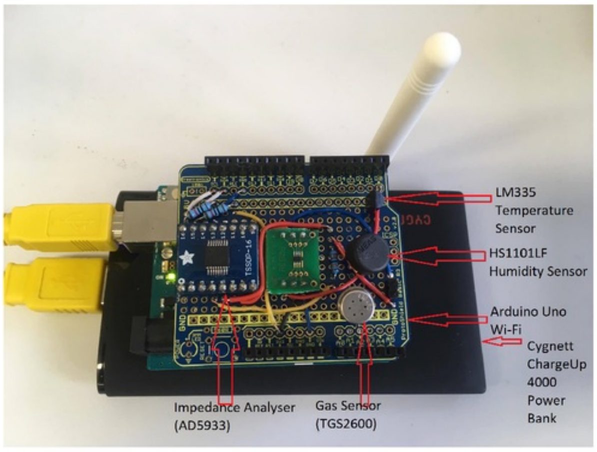


Figure 4.2: IoT-enabled smart sensing system for environment monitoring

4.2 LM335 Temperature Sensor Measurements

4.2.1 Calibration

Generally, LM335 does not need any calibration. However, to ensure higher efficiency, there is a pot connected across LM335 with an arm tied to the adjustment pin (pin 1) which allows a one-point calibration of the sensor. This calibration corrects any inaccuracy in the temperature readings over the full temperature range. This single-point calibration is meaningful since LM335 sensor output is proportional to absolute temperature with the extrapolated sensor output going to 0 volt at 0°Kelvin (-273°Celsius). Hence any error in output voltage versus temperature is just a slope, therefore, a calibration of this slope at one particular temperature rectifies errors at all temperatures. The temperature sensors voltage output is given by the following equation [17]:

$$Vout = \frac{VoutT0 \times T}{T0} \tag{4.1}$$

Where, T is the unknown temperature and T0 is the known reference temperature in °Kelvin. The output is usually calibrated at 10 mV/°Kelvin, therefore, at 25°Celsius or 298 °Kelvin, VoutT0 = $298°K \times 10mV/°K$ = 2.98V.
Thus the dropped voltage between the supply voltage (5V) and output voltage is = (5-2.98) V = 2.02V. Since this sensor is basically a Zener diode, a bias current must be generated in order to use it. According to the datasheet this current should be between 400 uA and 5 mA. A 2KΩ resistor has been used in the circuit to get this current.

4.2.2 Circuit Diagram and Temperature Results

The circuit connection for this sensor is very simple. The adjustable pin will remain unconnected and pin 2 and 3 will be connected to analog input 0 (A0) and the ground of the Arduino board respectively. A 2KΩ resistor will also be connected in series with pin 2 and 5V of the board. The circuit diagram and a screenshot of the code are given below:

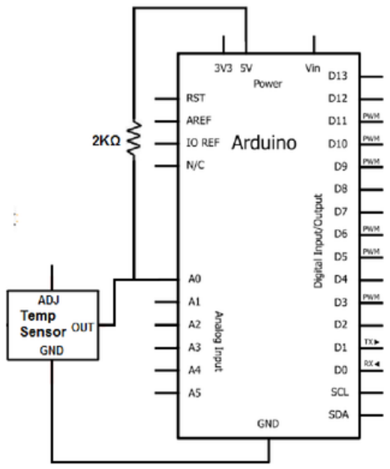
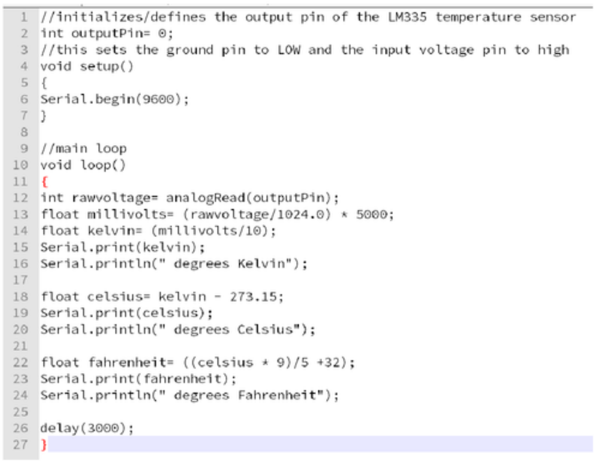


Figure 4.3: Circuit diagram of LM335 Temperature Sensor

According to the code, the analog output voltage or the raw voltage at A0 pin of the board is read first. Now this voltage is divided by 1024 since the span of 1024 holds the supply voltage 5V. The ratio of this raw voltage to the full span of 1024 is now achieved by multiplying it with 5000 in order to get the value in millivolts. Since 1024 represents the



```
//initializes/defines the output pin of the LM335 temperature sensor
int outputPin= 0;
//this sets the ground pin to LOW and the input voltage pin to high
void setup()
{
Serial.begin(9600);
}

//main loop
void loop()
{
int rawvoltage= analogRead(outputPin);
float millivolts= (rawvoltage/1024.0) * 5000;
float kelvin= (millivolts/10);
Serial.print(kelvin);
Serial.println(" degrees Kelvin");

float celsius= kelvin - 273.15;
Serial.print(celsius);
Serial.println(" degrees Celsius");

float fahrenheit= ((celsius * 9)/5 +32);
Serial.print(fahrenheit);
Serial.println(" degrees Fahrenheit");

delay(3000);
}
```

Figure 4.4: Code for LM335 Temperature Sensor

maximum value of the supply voltage (5V), the value of the raw voltage cannot be greater than 1024. The reason for multiplying the ratio with 5000 is it represents 5000 millivolts. Once the raw voltage is calculated in millivolts, the temperature in Kelvin is obtained by dividing the millivolts by 10. In order to get the equivalent Celsius temperature value, 273.15 is subtracted from the Kelvin value. Finally, the Fahrenheit temperature is attained from the following equation

$$Fahrenheit = \frac{Celsius \times 9}{5 + 32} \tag{4.2}$$

4.3 HS1101LF Relative Humidity Sensor Measurements

4.3.1 Calibration

In order to attain humidity results from this sensor, it has to be calibrated at first using impedance analyser (AD5933). After that the unknown impedance of the sensor is obtained which is used to calculate the capacitance and eventually the relative humidity of the surrounding environment. According to Ad5933 functional block diagram in the datasheet, a feedback resistor (Rfb) is connected in series with pin 4 and 5 (Vin) and another calibration resistor (Rcal) is connected in series with pin 5 and 6 (Vout) of the impedance analyser to perform the calibration. Performing frequency sweep is another important step during calibration which will be discussed in the next subsection. The calibration test is conducted during each frequency sweep using software algorithm to observe if the achieved impedance results match the known physical calibration resistor

(Rcal) value. 200 KΩ resistors have been used as both Rcal and Rfb resistors in this project.

Performing Frequency Sweep

All the necessary calculations to get unknown impedance and finally the humidity and temperature need to be performed during a frequency sweep. AD5933 can operate in a wide range of frequency which is from 1 Hz to 100 KHz. Frequency sweep is fully performed and controlled via programming of the three parameters, which are: start frequency, frequency increment and number of increments [2].

Start Frequency:

Start frequency is a 24-bit word that is programmed at Register Address 0x82, 0x83 and 0x84 inside the on-board RAM. The following equation is the code for the start frequency that needs to be loaded on the start frequency register [2]:

$$StartFrequencyCode = (\frac{RequiredOutputStartFrequency}{(MCLK/4)}) \times 2^{27} \tag{4.3}$$

Here,

Required Output Start Frequency = User defined start frequency;
MCLK = 16 MHz.

After putting the required values into the equation, the value needs to be converted into hexadecimal and stored into Register Address 0x82, 0x83 and 0x84. For example, if the value is 0x0F5C28 then the user need to program 0x0F to Register address 0x82, 0x5C to address 0x83 and 0x28 to address 0x84 [2].

Frequency Increment:

This is also a 24-bit word programmed at Register Address 0x85, 0x86 and 0x87 in the on-board Ram. The code is given by [2]:

$$FrequencyIncrement = (\frac{RequiredFrequencyIncrement}{(MCLK/4)}) \times 2^{27} \tag{4.4}$$

Here,

Required Frequency Increment = User defined frequency resolution during the sweep;
MCLK = 16 MHz.

Conversion and storing of the output from the equation in the registers are similar to the start frequency.

Number of Frequency Increments:

This is a 9-bit word that is also programmed to the on-board RAM at Register address 0x88 and 0x89. This number refers to the number of frequency points in each sweep. The highest value that can be programmed as the number of points is 511 [2].

After all the three parameters are programmed, a start frequency sweep command is issued at Register address 0x80 and 0x81. The user controls each frequency increment during the sweep. After each frequency increment command is executed, one set of real data from address 0x94, 0x95 and one set of imaginary data from address 0x96, 0x97 are read. After the frequency sweep is completed, bit D3 in the status register is set to indicate the completion of the frequency sweep command [2]. According to the humidity sensor datasheet, the capacitance measurements need to be done at 10 KHz/V. However, after a few trial and errors, expected results have been achieved for a frequency sweep from 11300 Hz to 12400 Hz with an increment of 100 Hz in this project. The number of increments was defined to be 10.

Impedance Calculation

At each frequency point during the sweep, a DFT is calculated. The AD5933 DFT algorithm is given by the following formula [2]:

$$X(f) = \sum_{n=0}^{1023}(x(n)(cos(n) - jsin(n))) \tag{4.5}$$

where:
X(f) is the power in the signal at the Frequency Point f;
x(n) is the ADC output;
cos(n) and sin(n) are the sampled test vectors provided by the DDS core at the Frequency Point f.

For each frequency point, the multiplication is sampled over the span of 1024. The output is stored in two 16 bit registers in the form of twos complement as real and imaginary component [2].

Magnitude Calculation:

In order to get impedance, the magnitude of the DFT has to be calculated first for each frequency increment. The equation of calculating magnitude is given by the following equation [2]:

$$Magnitude = \sqrt{(R^2 + I^2)} \tag{4.6}$$

Where, R is the real and I is the imaginary value stored in register address 0x94, 0x95 and 0x96, 0x97 respectively in hexadecimal. These hexadecimal values are converted to decimal internally in AD5933 and the magnitude is calculated by providing software instructions through Arduino IDE.

Gain Factor Calculation:

Calculating gain factors using the magnitude obtained earlier is the second step. The equation of gain factor calculation is given below [2]:

$$GainFactor = (\frac{Admittance}{Code}) = \frac{(\frac{1}{Impedance})}{Magnitude} \tag{4.7}$$

Here,
Impedance = Calibration resistor value = 200 KΩ; ;
Output Excitation Voltage = 2 V peak-to-peak;
PGA Gain = 1.

The impedance mentioned in this equation refers to the reference or calibration resistor value which is defined by the user in the code. As expected, the gain factors achieved in this project formed a linearly increasing line graph and were proportional to the frequency increments which is shown in the graph below where x-axis indicates frequency and y-axis indicates gain factors.

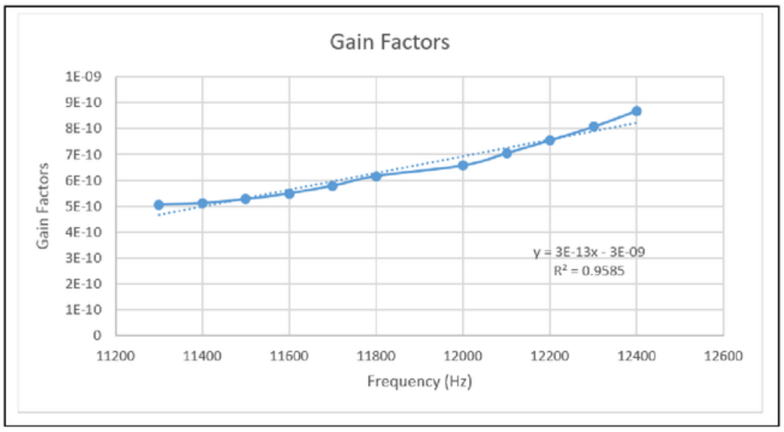


Figure 4.5: Frequency vs Gain Factors Chart

Impedance Calculation:

During calibration, the gain factors are used to calculate the impedance to ensure that the impedance value matches the value of the calibration resistor. The formula of calculating the impedance is given by [2]:

$$Impedance = \frac{1}{(GainFactor \times Magnitude)} \tag{4.8}$$

Here, the gain factor and magnitude have already been calculated in the previous two steps. All the gain factors and magnitudes have been calculated manually as well to ensure the software algorithm is working properly. The results were satisfactory and the impedance value received was 200KΩ which is the value of the calibration resistor. The gain factors are saved in an array for calculating unknown impedance in the next step.

4.3.2 Unknown Impedance Calculation

Once calibration test is successfully conducted, the next important step is to calculate the unknown impedance of the sensor. In this step, the calibration resistor would be removed and the sensor should be connected in that place. ADG849 mux will be used to complete the calibration automatically with the one run of the system and then move to calculating the unknown impedance. The formula is the same as impedance calculation, however, the gain factors would be the gain factor values attained from the calibration stage in this case. However, the magnitude will be calculated again using the sensor in the circuit to get new sets of real and imaginary values. The gain factors and the impedance values were averaged after performing three frequency sweeps within the same range to get more reliable data. The unknown impedance values were also proportional to the frequency increments according to the graph below. The x-axis represents frequency increments and the y-axis represent the averaged unknown impedance.

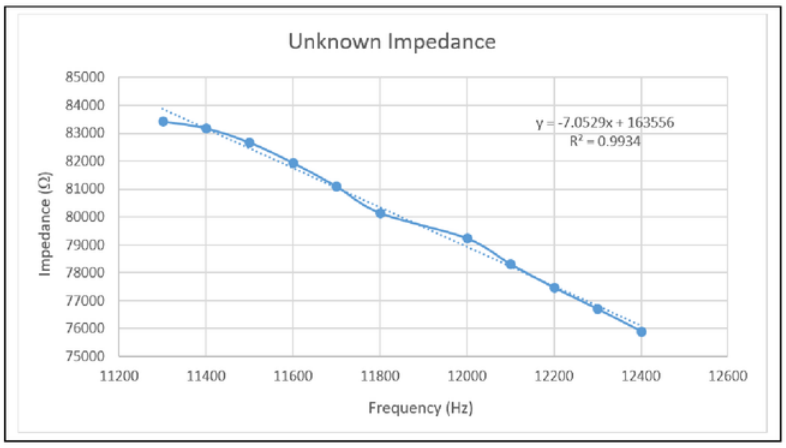


Figure 4.6: Frequency vs Unknown Impedance Chart

4.3.3 Phase Shift Calculation

Calculating phase shift is the next step in order to get the capacitance of the humidity sensor. The phase angle is calculated using the table provided in AD5933 datasheet which is given below [2]:

Real	Imaginary	Quadrant	Phase Angle
Positive	Positive	First	$tan^{-1}(\frac{I}{R}) \times \frac{180}{\pi}$
Negative	Positive	Second	$180 - (tan^{-1}(\frac{I}{R}) \times \frac{180}{\pi})$
Negative	Negative	Third	$180 + (tan^{-1}(\frac{I}{R}) \times \frac{180}{\pi})$
Positive	Negative	Fourth	$360 + (tan^{-1}(\frac{I}{R}) \times \frac{180}{\pi})$

Table 4.1: Phase Angle Calculation Table [2]

4.3.4 Humidity Results

After the phase shift has been calculated, the imaginary component of the sensor's impedance is obtained through the following equation:

$$ImaginaryImpedance = Impedance \times sin(phase) \tag{4.9}$$

Next, this imaginary impedance is used to calculate the capacitance using the following equation:

$$Capacitance = \frac{1}{2\pi \times frequency \times ImaginaryImpedance} \tag{4.10}$$

Where,
Frequency = frequency value at that point of frequency sweep.

The average value of capacitance has been taken in this project after performing the frequency sweep. Finally, this capacitance value is used to get the humidity according to the equation mentioned in the datasheet of humidity sensor (HS1101LF) [16]:

$$RH(\%) = -3.4656\times10^3\times X^3+1.0732\times10^4\times X^2-1.0457\times10^4\times X+3.2459\times10^3 \tag{4.11}$$

Where, RH (%) represents humidity in percentage and

$$X = C(read)/C@55\%RH \tag{4.12}$$

AD5933 impedance analyser also has a built-in 14-bit digital temperature sensor which is capable of measuring temperature with an accuracy of +/-2°Celsius within the measurement range. The temperature data are stored in register 0x92 and 0x93. The sensor has a temperature measurement range of -40°to 150°Celsius [2]. A function in the coding has been defined to get the temperature data and show it in Celsius value.

4.4 TGS2600 General Air Quality Sensor Measurements

This is an analog general air quality sensor which is capable of indicating concentration level of different gaseous contaminants in air, for example, it can detect up to 30 ppm of hydrogen gas in the air. It is generally used in the laboratories where different types of chemical gases are often used. As it has already been mentioned before, this sensor has four pins, the standard Vin and ground pins, a data pin and a separate Vin pin for the integrated heater. Both the Vin pins of the sensor and the heater are connected to the 5V pin of the Arduino board. The data pin is connected to analog input pin (A1) of the Arduino board. A 1 KΩ resistor is also connected in series with the data pin and the Ground of the Arduino. The basic measuring circuit of this sensor is given below [6]:

After the raw voltage of the analog input pin A1 has been read through software coding, the resistance of the sensor needs to be calculated according to the equation given in the TGS2600 sensor datasheet, which is [6]:

$$Rs = (\frac{Vc}{Vrl - 1}) \times Rl \tag{4.13}$$

Where,
Rs = Resistance of the sensor;
Vc = Supply voltage = 5V;
Vrl = output voltage = raw voltage*(5V/1024);
Rl = Load resistor = 1 KΩ.

According to the sensor's datasheet, the sensor's resistance should be in the range of 10K∼ 90KΩ. It has also provided a sensitivity characteristics graph for different types of gases, such as-hydrogen, ethanol, iso-butane, methane and carbon monoxide. The graph is shown below:

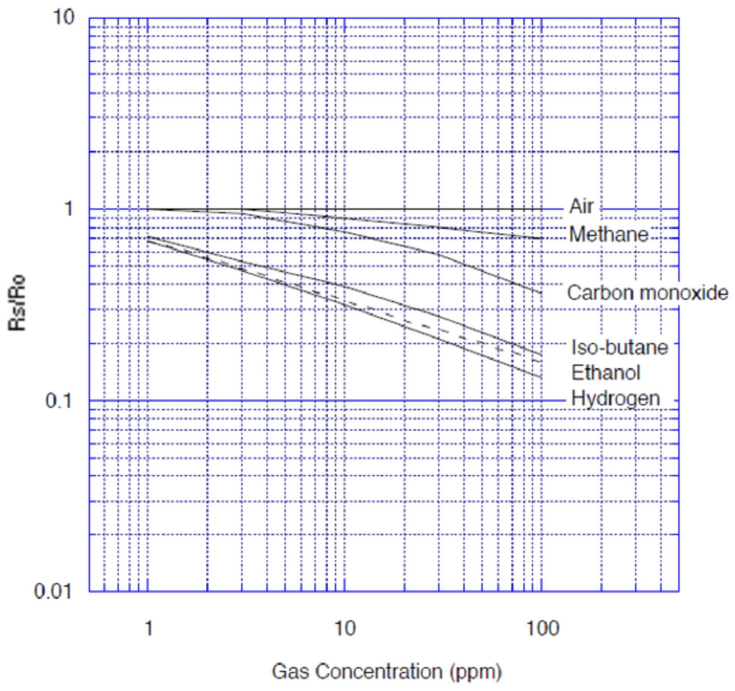
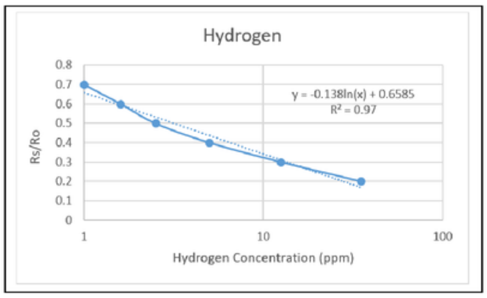
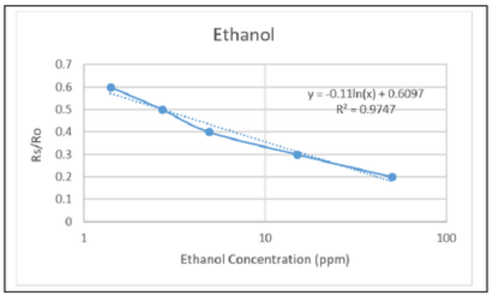


Figure 4.7: Sensitivity Graph for TGS2600 Module [6]

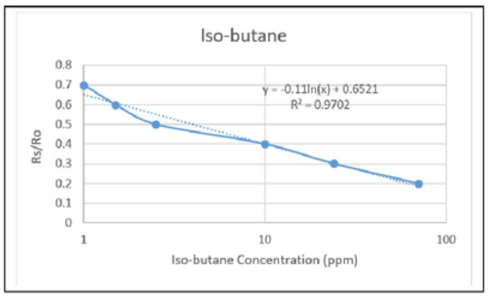
From this graph, distinct charts have been made using Microsoft Excel for each of the gases which are given below. These charts have provided distinct logarithmic equations for each of the gases which were written in the code in order to get the concentration level (in ppm) of each of the gaseous contaminant.



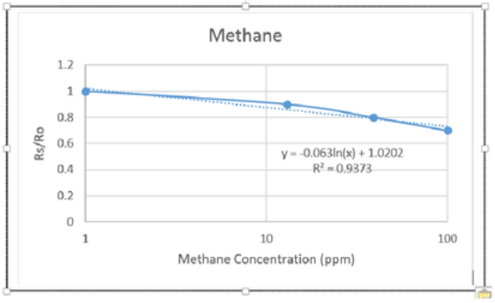
(a) Sensitivity characteristic graph for Hydrogen Gas



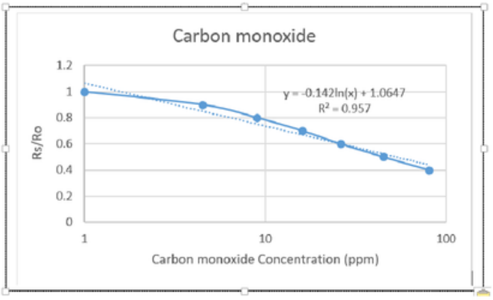
(b) Sensitivity characteristic graph for Ethanol Gas



(c) Sensitivity characteristic graph for Iso-butane Gas



(d) Sensitivity characteristic graph for Methane Gas



(e) Sensitivity characteristic graph for Carbon monoxide Gas

Figure 4.8: Rs/Ro Vs different level of gas concentration in ppm

The figures (4.8) in the previous page represent Rs/Ro versus concentration level in ppm of different gaseous contaminants [6] (hydrogen, ethanol, iso-butane, methane and carbon monoxide). Here,

Rs = Sensor resistance in displayed gases at various concentrations and
Ro = Sensor resistance in fresh air.

This chapter has illustrated all the theories and working procedures which were followed to develop the smart sensing system. The equations and graphs provided in this chapter were applied in the software coding to acquire correct results. The equations and graphs for calculating magnitude, gain factors and unknown impedance of the humidity sensor were obtained from the AD5933 datasheet. The equation to calculate humidity was provided in HS1101LF sensor datasheet. All the sensitivity characteristic graphs for the TGS2600 sensor were produced using the graph given in TGS2600 sensor datasheet.

Chapter 5

Results and Discussion

This chapter illustrates the results achieved in the field experiment by following the methodology explained in the previous chapter. The street view of the location of the field experiment is provided. The sensor data is sent directly to the ThingSpeak channel for monitoring. the screenshots of the channels are given and the results are explained. Visualisation charts using the imported data from the server have also been presented for further demonstration. Finally, the challenges encountered during the course of the project are explored in the discussion section.

5.1 Field Experiment Data

After processing all the sensor data according to the working procedures mentioned in the previous chapter, they are sent to the private Thingspeak channel through Wi-Fi connection. For field experiment, two similar sensor nodes were developed. Two private channels in the ThingSpeak server were created to update sensor data from those nodes separately for the convenience of monitoring them. The nodes were placed at two different spots inside Macquarie University campus. A screenshot of the location is given in the next page:

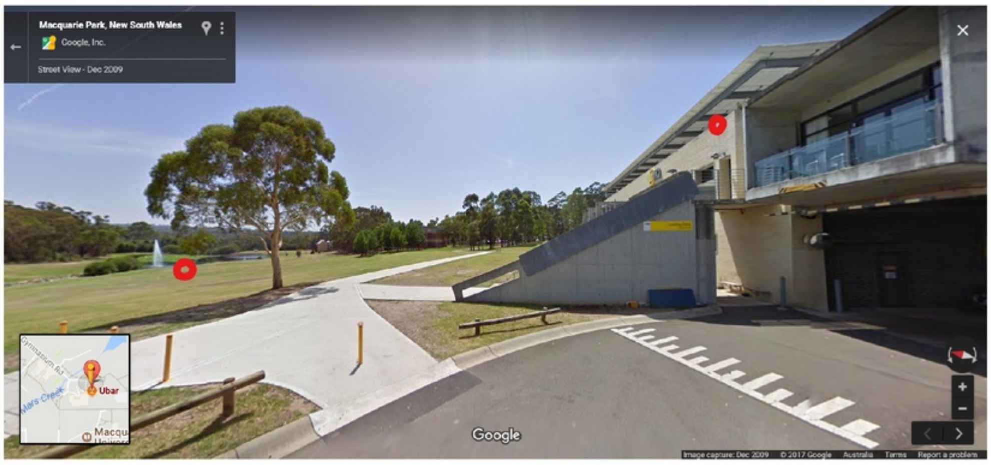

Macquarie Park, New South Wales
Google, Inc.
Street View - Dec 2009
Ubar
Google
Image capture: Dec 2009 © 2017 Google Australia Terms Report a problem

Figure 5.1: Location of the Sensor Nodes (Ubar, Macquarie University)

The red circles in the figure above indicate the exact locations of the sensor nodes. The system was on the run for two hours continuosuly to collect the field data and update them in the channels. The results were compared to the actual data from Google. Google shows the average data measured from a larger area, however, only two nodes were applied to measure data from two distinct locations in this case. Therefore, the data collected from these nodes were accurate precisely for those locations. The code was implemented in a way that each of the field was updated after an interval of 40 seconds. Screenshots of the updated channels are provided in the next page:

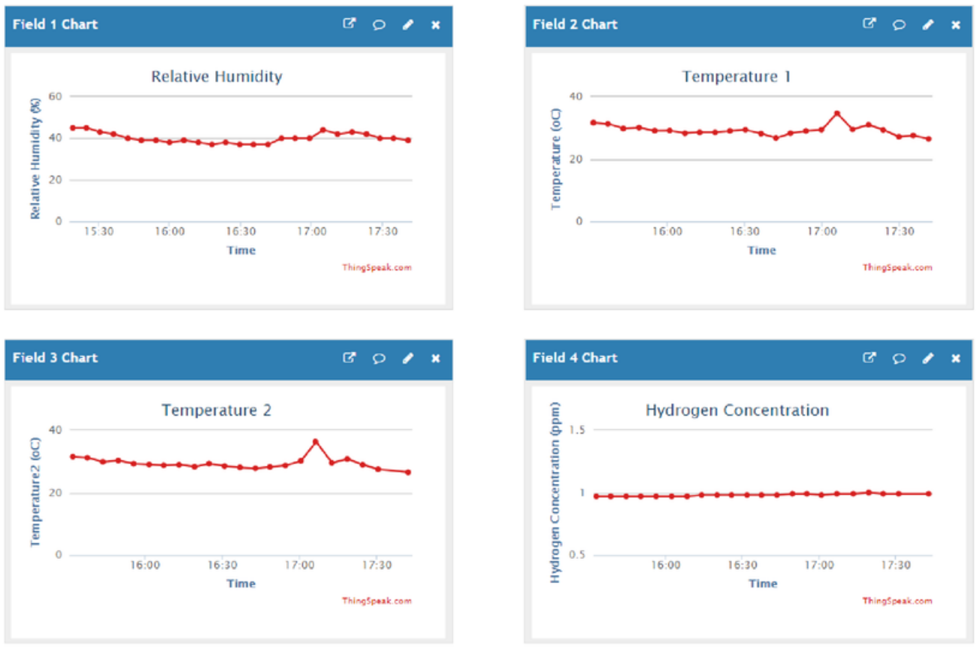


Figure 5.2: Field 1-4 with sensor data in ThingSpeak Channel 1

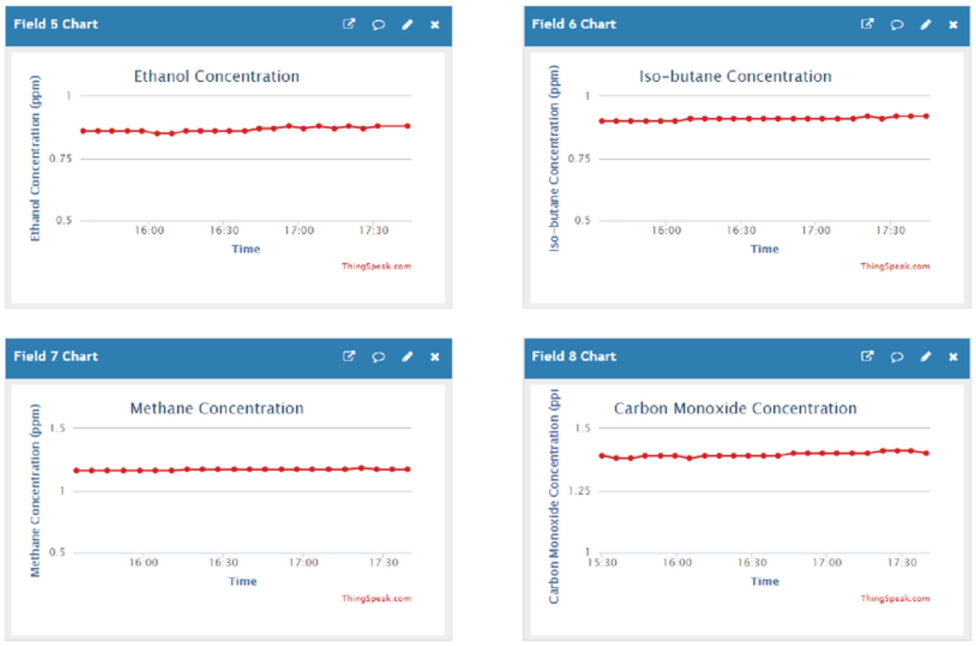


Figure 5.3: Field 5-8 with sensor data in ThingSpeak channel 1

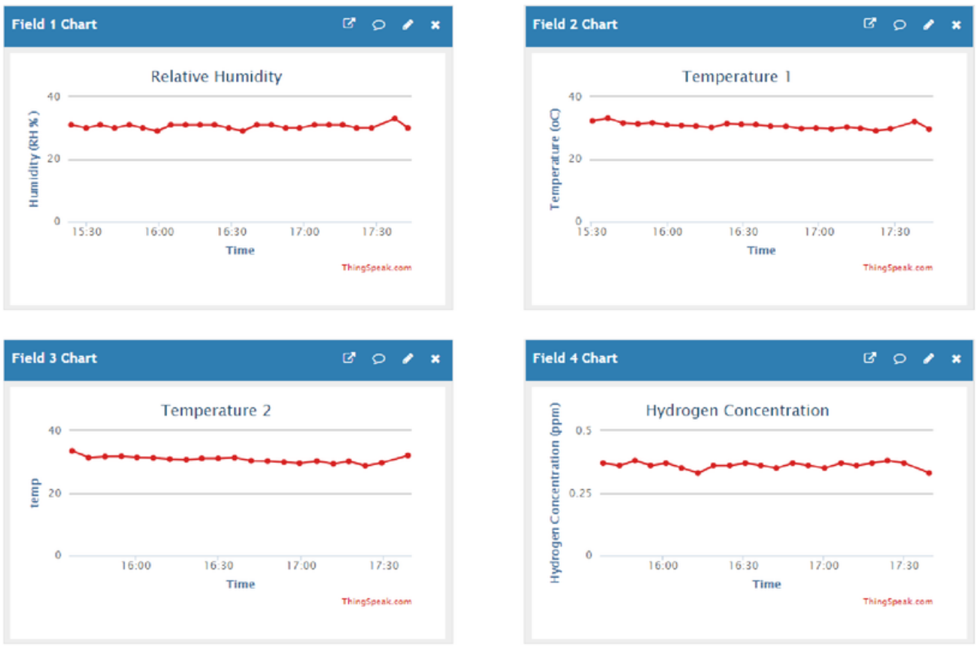


Figure 5.4: Field 1-4 with sensor data in ThingSpeak Channel 2

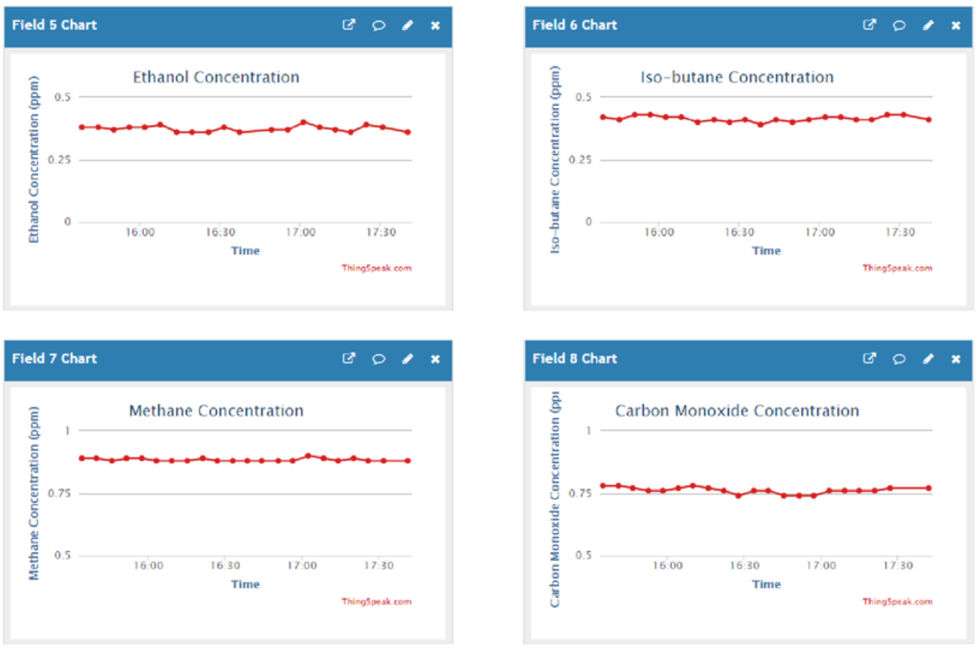
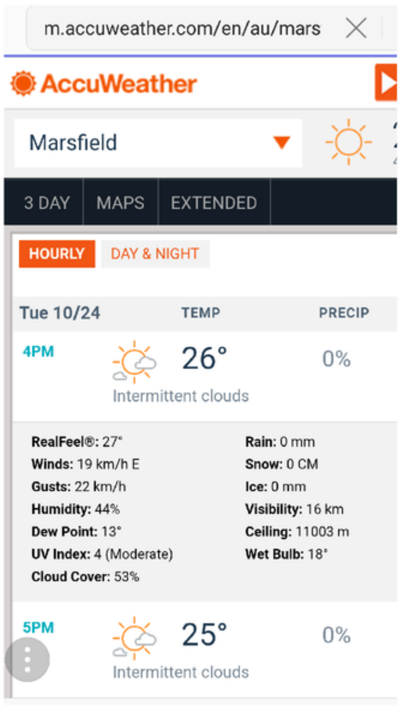


Figure 5.5: Field 5-8 with sensor data in ThingSpeak channel 2


m.accuweather.com/en/au/mars

AccuWeather

Marsfield

3 DAY | MAPS | EXTENDED

HOURLY | DAY & NIGHT

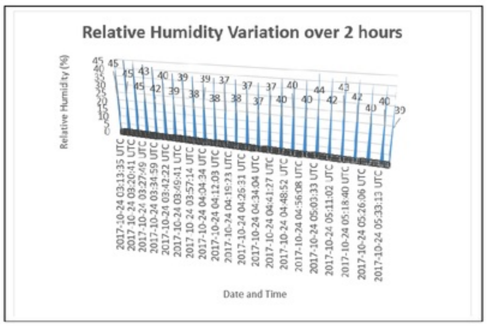
Tue 10/24	TEMP	PRECIP
4PM	26° Intermittent clouds	0%

RealFeel®: 27°	**Rain:** 0 mm
Winds: 19 km/h E	**Snow:** 0 CM
Gusts: 22 km/h	**Ice:** 0 mm
Humidity: 44%	**Visibility:** 16 km
Dew Point: 13°	**Ceiling:** 11003 m
UV Index: 4 (Moderate)	**Wet Bulb:** 18°
Cloud Cover: 53%	

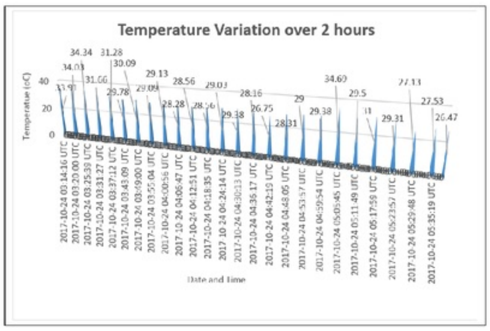
5PM	25° Intermittent clouds	0%

Figure 5.6: Exact Humidity and Temperature data taken from Google [7]

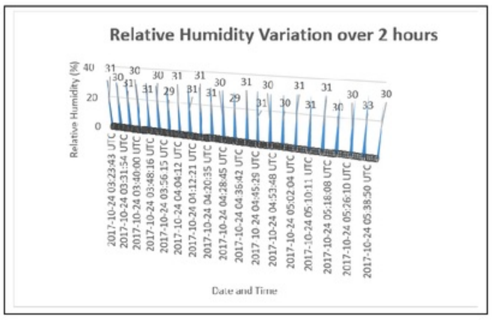
As mentioned above, two separate channels were created in ThingSpeak server for two similar nodes. Each channel contains eight fields for displaying humidity, temperature data from LM335 sensor, temperature data from AD5933 impedance analyser, hydrogen concentration, ethanol concentration, iso-butane concentration, methane concentration and carbon monoxide concentration respectively in the y axis and the x axis represent time in all the fields. The field experiment was conducted for two hours on Tuesday, 24 October 2017 from 3.30 pm to 5.30 pm. Figure 5.2 and 5.3 are referring to the eight fields of the first channel which are displaying data collected from the sensor node placed on the rooftop of the Ubar at Macquarie University campus. On the other hand, figure 5.4 and 5.5 represent the eight fields of the second channel displaying data from the other sensor placed outdoor. As it can be seen in case of the first channel, the humidity range was (39-45)% with reasonable fluctuations whereas, the second channel shows that the relative humidity was in between (30-31)% during those two hours. Field 2 and 3 of both channels show similar temperature range (26-34°Celsius) demonstrating consistency and accuracy of both temperature sensors. Since the sensor nodes were placed at a no smoking and any sort of chemical gas free area, the gas concentration level was very low with a range of only (0.86-1.4)ppm. Figure 5.6 shows the actual humidity and temperature data at 4pm on the same day obtained from "Accuweather" website via google search engine to compare with the experiment results. So far, the results achieved were almost 95% accurate.



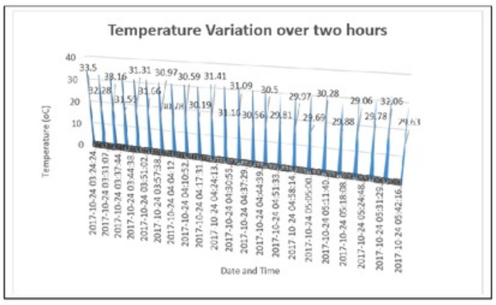
(a) Relative Humidity Variation over 2 hours achieved from first channel



(b) Temperature Variation over 2 hours achieved from first channel



(c) Relative Humidity Variation over 2 hours achieved from second channel



(d) Temperature Variation over 2 hours achieved from second channel

Figure 5.7: Relative humidity and temperature Variation charts obtained from imported ThingSpeak channel data

ThingSpeak being a very user-oriented platform, it also provides option for importing data from the channels for further analysis and demonstrations. Imported data can be used to create various types of visualisation charts through Microsoft Excel or Matlab coding. Figure (a) and (b) of 5.7 in the privious page represent visualisation charts of relative humidity and temperature variation over the 2 hours of field experiment which were attained from the imported data from ThingSpeak channel 1. Similarly, Figure (c) and (d) refer to the humidity and temperature variation charts obtained from imported data from ThingSpeak Channel 2. The x-axis in all the figures represents the date and time whereas the y-axis in figure (a) and (c) represents humidity and in figure (b) and (d) it indicates temperature at that time. From these figures, specific humidity and temperature data at specific date and time can be easily observed. Furthermore, the average, minimum or maximum value of these sensor data can easily be calculated by importing data from the channels.

5.2 Discussion

This section illustrates some of the problems encountered during the course of the project and their solutions-

5.2.1 Determining the Values of Calibration and Feedback Resistors

According to the theory explained in AD5933 datasheet, the values of the calibration and feedback resistors are calculated using the equation mentioned below [2]:

$$Rfb = \frac{\frac{VDD}{2} - 0.2 \times Zmin}{Vpk + \frac{VDD}{2} - Vdcoffset} \times \frac{1}{Gain} \tag{5.1}$$

Here,
Rfb = feedback resistor;
VDD = supply voltage = 3.3 V;
Zmin = minimum impedance range;
Vpk = peak voltage of the selected output range = 2 V peak-to-peak;
Gain = PGA gain = 1;
Vdcoffset = the DC offset voltage for the selected range = 1.48 V (from the datasheet).

$$Rcal = (Zmin + Zmax) \times \frac{1}{3} \tag{5.2}$$

Where,
Rcal = calibration resistor;
Zmin = minimum impedance range;
Zmax = maximum impedance range.

According to these equations, the calibration and feedback resistors were calculated for the impedance ranges given below:

Zmin KΩ	Zmax KΩ	Rfb KΩ	Rcal KΩ
0.1	1	0.07667	0.366
1	10	0.7667	3.67
10	100	7.667	36.67
100	1000	76.67	366.67

Table 5.1: Calibration and feedback resistor values for different ranges of impedance

However, the impedances received for each set of these values in the calibration stage did not match with value of calibration resistor. After several attempts, the co-supervisor suggested to use 200 KΩ for both the resistors and the results were satisfactory.

5.2.2 Stability and Storing Issues of Gain Factors

The gain factors are expected to represent a linear curve or line while plotted against frequency. However, random changes in the gain factors were noticed in the beginning of the project. After performing an experiment in the lab, it became evident that resistors show a little change in their resistance values as well as become a little capacitive in high frequency whereas pure resistive elements should have been used for this experiment. Therefore, a low frequency range has been decided (11300-12400 Hz) to perform the frequency sweep until this stage of the project.

Storing the gain factors in an array was another issue that was encountered during the project. Arduino Uno Wi-Fi has a very low memory (32 KB) and does not contain any SD card slot. Since it cannot store a large amount of data, the number of frequency increments has been kept low for now. This issue has not been resolved yet.

Chapter 6

Conclusion and Future Work

To summarise the whole project, the goal of this project was to establish an innovative low-cost sensor node containing both resistive and capacitive sensors and send the processed data to an IoT server for environment monitoring. For this purpose, a capacitive humidity sensor and two resistive temperature and gas sensors were used in one connected circuit. According to the research conducted for this project, this type of sensor node is rarely available in the market due to the complexity of the development process. Chapter 2 has illustrated the literature review and background knowledge of this particular project. The detailed system description as well as system components are mentioned and described in chapter 3.

Chapter 4 discusses all the necessary working principles for the three sensors used in this project. LM335 is a simple and accurate temperature sensor. Since this is a resistive sensor, it could be directly interfaced to the Arduino microcontroller. Its datapin was connected to one of the analog inputs of the Arduino board to get the raw voltage of the input pin. This voltage is then used to measure the temperature using necessary software coding which has been explained in chapter 4.

AD5933 impedance analyser was introduced to create communication between HS1101LF humidity sensor and the main microcontroller. At first the sensor was calibrated by performing a frequency sweep which is dependent on three variables− start frequency, frequency increment and the number of increments. These variables are defined by the user according to the characteristics of the sensor. In this project, the frequency sweep range was 11,300 Hz to 12,400 Hz with an increment of 100 Hz at each point. The number of increments was defined as 10. A calibration resistor and a feedback resistor of same value (200 kΩ) were used for the calibration test. The next step was to obtain the real and imaginary component from the respective register addresses inside the on-board RAM of the impedance analyser in order to calculate the magnitude at frequency point. After that, these magnitudes and the user defined reference resistance value were used to calculate the impedance to ensure that the impedance value and reference resistance value matched.

When the calibration test was successful, the calibration resistor was removed and the sensor was placed instead to measure its unknown impedance. The gain factors achieved in the previous stage were used in this case to calculate the impedance where the equation remained the same. However, the magnitudes were different since the real and imaginary components should be different for the sensor. After gaining the impedance values for the sensor, they were averaged after every three frequency sweep in order to get more accurate results. These impedance values are required to calculate phase shift and capacitance which is the most important phase to get humidity results. Humidity is calculated using the equation given in the HS1101LF sensor datasheet. Expected results were achieved following the above steps with almost 95% of accuracy.

TGS2600 gas sensor is a general air quality sensor which is usually required in the laboratories where harmful chemical gases are used. It can detect almost 1∼30 ppm of hydrogen gas and has a high sensitivity to low gas concentration levels only. The gas concentration level detection principle depends on the sensor resistance in that environment as well as in fresh air. Equations were developed for each gas using the sensitivity characteristic graph given in the datasheet. However, practical experiment need to be conducted to get the accurate sensor resistance value in fresh air and in the environment where the gas concentration level will be measured.

Once expected results were attained for all the three sensors, the next step was to establish an IoT configuration to make the whole system a connected end-to-end arrangement. This has been discussed in detail in chapter 5. ThingSpeak was chosen as the IoT platform for this project which is a user-friendly open-source application. Two similar nodes were developed to make it a smart sensor network and the sensor data from those nodes were updated successfully to two different private channels in the ThingSpeak server. In short, it can be said that all the milestones were achieved with expected results making it an overall successful project.

For future work, LoRa shield, which stands for ”long range”, can be introduced instead of the Arduino’s Wi-Fi shield for its capacity of covering larger distance (5-10 kilometers). Due to time constraint, an elaborate analysis and demonstration could not be carried out in this project. Further Matlab analysis can also be done in the ThingSpeak server using the sensor data for more advanced demonstrations according to the user’s requirements. Only two sensor nodes have been developed until now to make it a sensor network. However, larger number of sensor nodes would be able to provide more accurate results making the system more reliable for environment monitroing.

Appendix A

Project Timeline and Consultation Attendance Form

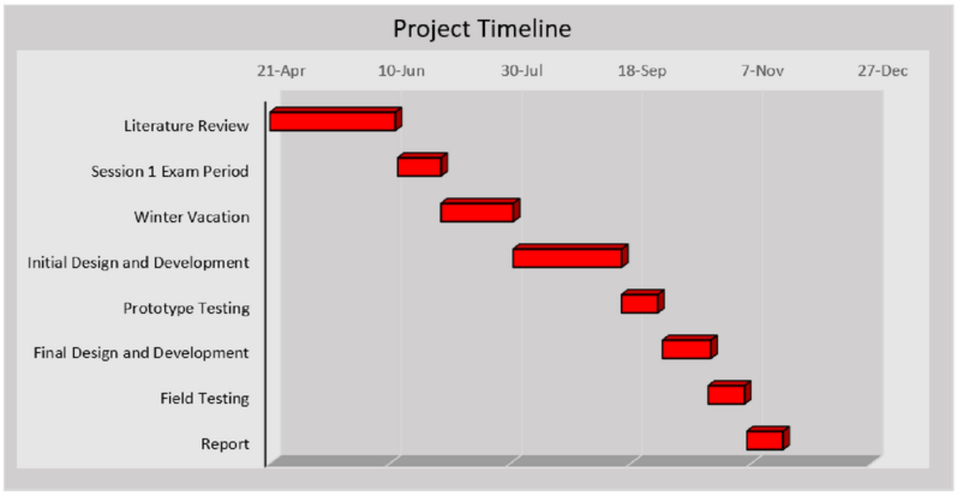
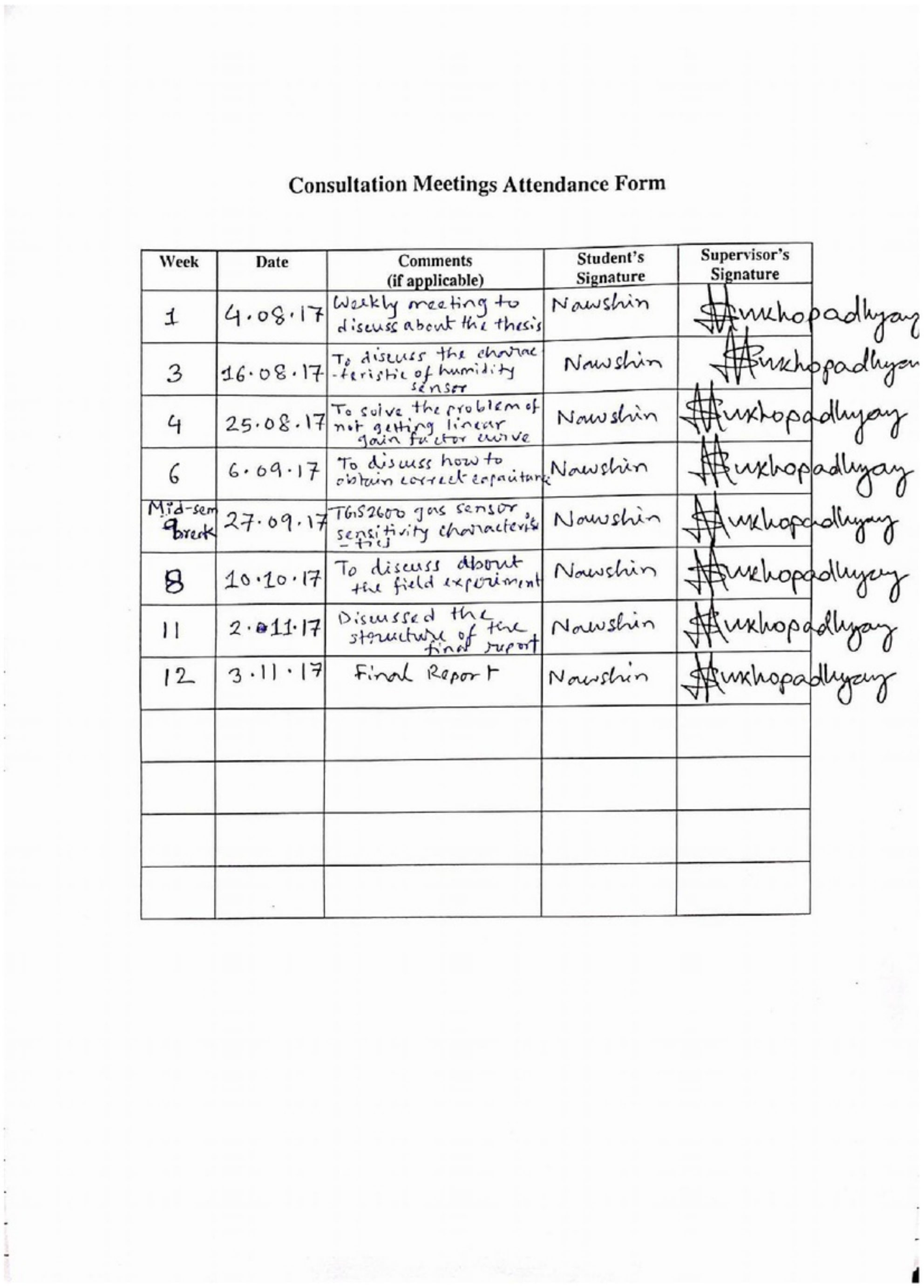


Figure A.1: Gantt chart of the Project Timeline



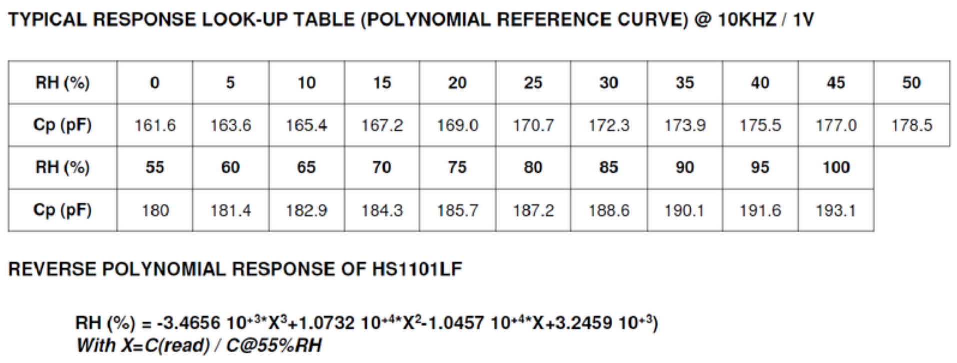
Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
1	4.08.17	Weekly meeting to discuss about the thesis	Nawshin	[illegible]
3	16.08.17	To discuss the charac-teristic of humidity sensor	Nawshin	[illegible]
4	25.08.17	To solve the problem of not getting linear gain factor curve	Nawshin	[illegible]
6	6.09.17	To discuss how to obtain correct capacitance	Nawshin	[illegible]
Mid-sem break 9	27.09.17	TGS2600 gas sensor, sensitivity characteris-tics	Nawshin	[illegible]
8	10.10.17	To discuss about the field experiment	Nawshin	[illegible]
11	2.11.17	Discussed the structure of the final report	Nawshin	[illegible]
12	3.11.17	Final Report	Nawshin	[illegible]

Figure A.2: Consultation attendance Form

Appendix B

LookUp Table for Humidity Sensor



TYPICAL RESPONSE LOOK-UP TABLE (POLYNOMIAL REFERENCE CURVE) @ 10KHZ / 1V

RH (%)	**0**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
Cp (pF)	161.6	163.6	165.4	167.2	169.0	170.7	172.3	173.9	175.5	177.0	178.5
RH (%)	**55**	**60**	**65**	**70**	**75**	**80**	**85**	**90**	**95**	**100**	
Cp (pF)	180	181.4	182.9	184.3	185.7	187.2	188.6	190.1	191.6	193.1	

REVERSE POLYNOMIAL RESPONSE OF HS1101LF

RH (%) = -3.4656 10^{+3}*X^3+1.0732 10^{+4}*X^2-1.0457 10^{+4}*X+3.2459 10^{+3})
With X=C(read) / C@55%RH

Figure B.1: Lookup table for humidity calculation given in the datasheet

Appendix C

Arduino Codes developed for Smart Sensing System

C.1 Final code for measuring sensor data and sending them to ThingSpeak Channel

C.1.1 AD5933 Header File

```
#ifndef AD5933_h
#define AD5933_h

/**
 * Includes
 */
#include <Arduino.h>
#include <Wire.h>

/**
 * AD5933 Register Map
 *  Datasheet p23
 */
// Device address and address pointer
#define AD5933_ADDR     (0x0D)
#define ADDR_PTR        (0xB0)
// Control Register
#define CTRL_REG1       (0x80)
#define CTRL_REG2       (0x81)
// Start Frequency Register
#define START_FREQ_1    (0x82)
#define START_FREQ_2    (0x83)
```

```
#define START_FREQ_3    (0x84)
// Frequency increment register
#define INC_FREQ_1      (0x85)
#define INC_FREQ_2      (0x86)
#define INC_FREQ_3      (0x87)
// Number of increments register
#define NUM_INC_1       (0x88)
#define NUM_INC_2       (0x89)
// Number of settling time cycles register
#define NUM_SCYCLES_1   (0x8A)
#define NUM_SCYCLES_2   (0x8B)
// Status register
#define STATUS_REG      (0x8F)
// Temperature data register
#define TEMP_DATA_1     (0x92)
#define TEMP_DATA_2     (0x93)
// Real data register
#define REAL_DATA_1     (0x94)
#define REAL_DATA_2     (0x95)
// Imaginary data register
#define IMAG_DATA_1     (0x96)
#define IMAG_DATA_2     (0x97)

/**
 * Constants
 *  Constants for use with the AD5933 library class.
 */
// Temperature measuring
#define TEMP_MEASURE    (CTRL_TEMP_MEASURE)
#define TEMP_NO_MEASURE (CTRL_NO_OPERATION)
// Clock sources
#define CLOCK_INTERNAL  (CTRL_CLOCK_INTERNAL)
#define CLOCK_EXTERNAL  (CTRL_CLOCK_EXTERNAL)
// PGA gain options
#define PGA_GAIN_X1     (CTRL_PGA_GAIN_X1)
#define PGA_GAIN_X5     (CTRL_PGA_GAIN_X5)
// Power modes
#define POWER_STANDBY   (CTRL_STANDBY_MODE)
#define POWER_DOWN      (CTRL_POWER_DOWN_MODE)
#define POWER_ON        (CTRL_NO_OPERATION)
// I2C result success/fail
#define I2C_RESULT_SUCCESS       (0)
#define I2C_RESULT_DATA_TOO_LONG (1)
```

```
#define I2C_RESULT_ADDR_NAK      (2)
#define I2C_RESULT_DATA_NAK      (3)
#define I2C_RESULT_OTHER_FAIL    (4)
// Control register options
#define CTRL_NO_OPERATION       (0b00000000)
#define CTRL_INIT_START_FREQ    (0b00010000)
#define CTRL_START_FREQ_SWEEP   (0b00100000)
#define CTRL_INCREMENT_FREQ     (0b00110000)
#define CTRL_REPEAT_FREQ        (0b01000000)
#define CTRL_TEMP_MEASURE       (0b10010000)
#define CTRL_POWER_DOWN_MODE    (0b10100000)
#define CTRL_STANDBY_MODE       (0b10110000)
#define CTRL_RESET              (0b00010000)
#define CTRL_CLOCK_EXTERNAL     (0b00001000)
#define CTRL_CLOCK_INTERNAL     (0b00000000)
#define CTRL_PGA_GAIN_X1        (0b00000001)
#define CTRL_PGA_GAIN_X5        (0b00000000)
// Status register options
#define STATUS_TEMP_VALID       (0x01)
#define STATUS_DATA_VALID       (0x02)
#define STATUS_SWEEP_DONE       (0x04)
#define STATUS_ERROR            (0xFF)
// Frequency sweep parameters
#define SWEEP_DELAY             (1)

/**
 * AD5933 Library class
 *  Contains mainly functions for interfacing with the AD5933.
 */
class AD5933 {
    public:
        // Reset the board
        static bool reset(void);

        // Temperature measuring
        static bool enableTemperature(byte);
        static double getTemperature(void);

        // Clock
        static bool setClockSource(byte);
        static bool setInternalClock(bool);
        //bool setSettlingCycles(int); // not implemented - not
            used yet

        // Frequency sweep configuration
        static bool setStartFrequency(unsigned long);
        static bool setIncrementFrequency(unsigned long);
        static bool setNumberIncrements(unsigned int);

        // Gain configuration
        static bool setPGAGain(byte);

        // Excitation range configuration
        //bool setRange(byte, int); // not implemented - not used yet

        // Read registers
        static byte readRegister(byte);
        static byte readStatusRegister(void);
        static int readControlRegister(void);

        // Impedance data
        static bool getComplexData(int*, int*);

        // Set control mode register (CTRL_REG1)
        static bool setControlMode(byte);

        // Power mode
        static bool setPowerMode(byte);

        // Perform frequency sweeps
        static bool frequencySweep(int real[], int imag[], int);
        static bool calibrate(double gain[], int phase[], int ref, int n);
        static bool calibrate(double gain[], int phase[], int real[],
                              int imag[], int ref, int n);
        static float GetHumidity ();
    private:
        // Private data
        static const unsigned int clockSpeed = 16776000;

        // Sending/Receiving byte method, for easy re-use
        static int getByte(byte, byte*);
        static bool sendByte(byte, byte);
```

```
};

#endif
```

C.1.2 AD5933 Source Code

```
/**
 * @file AD5933.cpp
 * @brief Library code for AD5933
 */

#include "AD5933.h"
#include <Math.h>
#define START_FREQ  (11300)
#define FREQ_INCR   (100)
#define NUM_INCR    (10)
#define REF_RESIST  (201200)

/**
 * Request to read a byte from the AD5933.
 *
 * @param address Address of register requesting data from
 * @param value Pointer to a byte where the return value should
     be stored, or
 *        where the error code will be stored if fail.
 * @return Success or failure
 */
int AD5933::getByte(byte address, byte *value) {
    // Request to read a byte using the address pointer
       register
    Wire.beginTransmission(AD5933_ADDR);
    Wire.write(ADDR_PTR);
    Wire.write(address);

    // Ensure transmission worked
    if (byte res = Wire.endTransmission() != I2C_RESULT_SUCCESS
       ) {
        *value = res;
        return false;
    }

    // Read the byte from the written address
    Wire.requestFrom(AD5933_ADDR, 1);
    if (Wire.available()) {
        *value = Wire.read();
        return true;
    } else {
        *value = 0;
        return false;
    }
}

/**
 * Write a byte to a register on the AD5933.
 *
 * @param address The register address to write to
 * @param value The byte to write to the address
 * @return Success or failure of transmission
 */
bool AD5933::sendByte(byte address, byte value) {
    // Send byte to address
    Wire.beginTransmission(AD5933_ADDR);
    Wire.write(address);
    Wire.write(value);

    // Check that transmission completed successfully
    if (byte res = Wire.endTransmission() != I2C_RESULT_SUCCESS
       ) {
        return false;
    } else {
        return true;
    }
}

/**
 * Set the control mode register, CTRL_REG1. This is the
   register where the
 * current command needs to be written to so this is used a lot
   .
 *
 * @param mode The control mode to set
 * @return Success or failure
 */
bool AD5933::setControlMode(byte mode) {
    // Get the current value of the control register
    byte val;
    if (!getByte(CTRL_REG1, &val))
```

```
        return false;

    // Wipe out the top 4 bits...mode bits are bits 5 through
       8.
    val &= 0x0F;

    // Set the top 4 bits appropriately
    val |= mode;

    // Write back to the register
    return sendByte(CTRL_REG1, val);
}

/**
 * Reset the AD5933. This interrupts a sweep if one is running,
     but the start
 * frequency, number of increments, and frequency increment
    register contents
 * are not overwritten, but an initialize start frequency
    command is required
 * to restart a frequency sweep.
 *
 * @return Success or failure
 */
bool AD5933::reset() {
    // Get the current value of the control register
    byte val;
    if (!getByte(CTRL_REG2, &val))
        return false;

    // Set bit D4 for restart
    val |= CTRL_RESET;

    // Send byte back
    return sendByte(CTRL_REG2, val);
}

/**
 * Set enable temperature measurement. This interferes with
    frequency sweep
 * operation, of course.
 *
 * @param enable Option to enable to disable temperature
```

```
    measurement.
 * @return Success or failure
 */
bool AD5933::enableTemperature(byte enable) {
    // If enable, set temp measure bits. If disable, reset to
       no operation.
    if (enable == TEMP_MEASURE) {
        return setControlMode(CTRL_TEMP_MEASURE);
    } else {
        return setControlMode(CTRL_NO_OPERATION);
    }
}

/**
 * Get the temperature reading from the AD5933. Waits until a
    temperature is
 * ready. Also ensures temperature measurement mode is active.
 *
 * @return The temperature in celcius, or -1 if fail.
 */
double AD5933::getTemperature() {
    // Set temperature mode
    if (enableTemperature(TEMP_MEASURE)) {
        // Wait for a valid temperature to be ready
        while((readStatusRegister() & STATUS_TEMP_VALID) !=
           STATUS_TEMP_VALID) ;

        // Read raw temperature from temperature registers
        byte rawTemp[2];
        if (getByte(TEMP_DATA_1, &rawTemp[0]) &&
            getByte(TEMP_DATA_2, &rawTemp[1]))
        {
            // Combine raw temperature bytes into an interger.
               The ADC
            // returns a 14-bit 2's C value where the 14th bit
               is a sign
            // bit. As such, we only need to keep the bottom 13
                bits.
            int rawTempVal = (rawTemp[0] << 8 | rawTemp[1]) & 0
               x1FFF;

            // Convert into celcius using the formula given in
               the
            // datasheet. There is a different formula
               depending on the sign
            // bit, which is the 5th bit of the byte in
               TEMP_DATA_1.
            if ((rawTemp[0] & (1<<5)) == 0) {
                return rawTempVal / 32.0;
            } else {
                return (rawTempVal - 16384) / 32.0;
            }
        }
    }
    return -1;
}


/**
 * Set the color source. Choices are between internal and
   external.
 *
 * @param source Internal or External clock
 * @return Success or failure
 */
bool AD5933::setClockSource(byte source) {
    // Determine what source was selected and set it
       appropriately
    switch (source) {
        case CLOCK_EXTERNAL:
            return sendByte(CTRL_REG2, CTRL_CLOCK_EXTERNAL);
        case CLOCK_INTERNAL:
            return sendByte(CTRL_REG2, CTRL_CLOCK_INTERNAL);
        default:
            return false;
    }
}

/**
 * Set the color source to internal or not.
 *
 * @param internal Whether or not to set the clock source as
   internal.
 * @return Success or failure
 */
bool AD5933::setInternalClock(bool internal) {
    // This function is mainly a wrapper for setClockSource()
    if (internal)
        return setClockSource(CLOCK_INTERNAL);
    else
        return setClockSource(CLOCK_EXTERNAL);
}

/**
 * Set the start frequency for a frequency sweep.
 *
 * @param start The initial frequency.
 * @return Success or failure
 */
bool AD5933::setStartFrequency(unsigned long start) {
    // Page 24 of the Datasheet gives the following formula to
       represent the
    // start frequency.
    // TODO: Precompute for better performance if we want to
       keep this constant.
    long freqHex = (start / (16776000 / 4.0))*pow(2, 27);
    if (freqHex > 0xFFFFFF) {
        return false;   // overflow
    }

    // freqHex should be a 24-bit value. We need to break it up
        into 3 bytes.
    byte highByte = (freqHex >> 16) & 0xFF;
    byte midByte = (freqHex >> 8) & 0xFF;
    byte lowByte = freqHex & 0xFF;

    // Attempt sending all three bytes
    return sendByte(START_FREQ_1, highByte) &&
           sendByte(START_FREQ_2, midByte) &&
           sendByte(START_FREQ_3, lowByte);
}

/**
 * Set the increment frequency for a frequency sweep.
 *
 * @param start The frequency to increment by. Max of 0xFFFFFF.
 * @return Success or failure
 */
bool AD5933::setIncrementFrequency(unsigned long increment) {
    // Page 25 of the Datasheet gives the following formula to
       represent the
    // increment frequency.
    // TODO: Precompute for better performance if we want to
       keep this constant.
    long freqHex = (increment / (16776000/ 4.0))*pow(2, 27);
    if (freqHex > 0xFFFFFF) {
        return false;   // overflow
    }

    // freqHex should be a 24-bit value. We need to break it up
        into 3 bytes.
    byte highByte = (freqHex >> 16) & 0xFF;
    byte midByte = (freqHex >> 8) & 0xFF;
    byte lowByte = freqHex & 0xFF;

    // Attempt sending all three bytes
    return sendByte(INC_FREQ_1, highByte) &&
           sendByte(INC_FREQ_2, midByte) &&
           sendByte(INC_FREQ_3, lowByte);
}

/**
 * Set the number of frequency increments for a frequency sweep
    .
 *
 * @param start The number of increments to use. Max 511.
 * @return Success or failure
 */
bool AD5933::setNumberIncrements(unsigned int num) {
    // Check that the number sent in is valid.
    if (num > 511) {
        return false;
    }

    // Divide the 9-bit integer into 2 bytes.
    byte highByte = (num >> 8) & 0xFF;
    byte lowByte = num & 0xFF;

    // Write to register.
    return sendByte(NUM_INC_1, highByte) &&
           sendByte(NUM_INC_2, lowByte);
}
```

```

/**
 * Set the PGA gain factor.
 *
 * @param gain The gain factor to select. Use constants or 1/5.
 * @return Success or failure
 */
bool AD5933::setPGAGain(byte gain) {
    // Get the current value of the control register
    byte val;
    if (!getByte(CTRL_REG1, &val))
        return false;

    // Clear out the bottom bit, D8, which is the PGA gain set
       bit
    val &= 0xFE;

    // Determine what gain factor was selected
    if (gain == PGA_GAIN_X1 || gain == 1) {
        // Set PGA gain to x1 in CTRL_REG1
        val |= PGA_GAIN_X1;
        return sendByte(CTRL_REG1, val);
    } else if (gain == PGA_GAIN_X5 || gain == 5) {
        // Set PGA gain to x5 in CTRL_REG1
        val |= PGA_GAIN_X5;
        return sendByte(CTRL_REG1, val);
    } else {
        return false;
    }
}

/**
 * Read the value of a register.
 *
 * @param reg The address of the register to read.
 * @return The value of the register. Returns 0xFF if can't
   read it.
 */
byte AD5933::readRegister(byte reg) {
    // Read status register and return it's value. If fail,
       return 0xFF.
    byte val;
    if (getByte(reg, &val)) {
        return val;
    } else {
        return STATUS_ERROR;
    }
}

/**
 * Read the value of the status register.
 *
 * @return The value of the status register. Returns 0xFF if
   can't read it.
 */
byte AD5933::readStatusRegister() {
    return readRegister(STATUS_REG);
}

/**
 * Read the value of the control register.
 *
 * @return The value of the control register. Returns 0xFFFF if
   can't read it.
 */
int AD5933::readControlRegister() {
    return ((readRegister(CTRL_REG1) << 8) | readRegister(
       CTRL_REG2)) & 0xFFFF;
}

/**
 * Get a raw complex number for a specific frequency
   measurement.
 *
 * @param real Pointer to an int that will contain the real
   component.
 * @param imag Pointer to an int that will contain the
   imaginary component.
 * @return Success or failure
 */
bool AD5933::getComplexData(int *real, int *imag) {
    // Wait for a measurement to be available
    while ((readStatusRegister() & STATUS_DATA_VALID) !=
       STATUS_DATA_VALID);

    // Read the four data registers.
    // TODO: Do this faster with a block read
    byte realComp[2];
    byte imagComp[2];
    if (getByte(REAL_DATA_1, &realComp[0]) &&
        getByte(REAL_DATA_2, &realComp[1]) &&
        getByte(IMAG_DATA_1, &imagComp[0]) &&
        getByte(IMAG_DATA_2, &imagComp[1]))
    {
        // Combine the two separate bytes into a single 16-bit
           value and store
        // them at the locations specified.
        *real = (int16_t)(((realComp[0] << 8) | realComp[1]) &
           0xFFFF);
        *imag = (int16_t)(((imagComp[0] << 8) | imagComp[1]) &
           0xFFFF);

        return true;
    } else {
        *real = -1;
        *imag = -1;
        return false;
    }
}

/**
 * Set the power level of the AD5933.
 *
 * @param level The power level to choose. Can be on, standby,
   or down.
 * @return Success or failure
 */
bool AD5933::setPowerMode(byte level) {
    // Make the appropriate switch. TODO: Does no operation
       even do anything?
    switch (level) {
        case POWER_ON:
            return setControlMode(CTRL_NO_OPERATION);
        case POWER_STANDBY:
            return setControlMode(CTRL_STANDBY_MODE);
        case POWER_DOWN:
            return setControlMode(CTRL_POWER_DOWN_MODE);
        default:
            return false;
    }
}

/**
 * Perform a complete frequency sweep.
 *
 * @param real An array of appropriate size to hold the real
    data.
 * @param imag An array of appropriate size to hold the
    imaginary data.
 * @param n Length of the array (or the number of discrete
    measurements)
 * @return Success or failure
 */
bool AD5933::frequencySweep(int real[], int imag[], int n) {
    // Begin by issuing a sequence of commands
    // If the commands aren't taking hold, add a brief delay
    if (!(setPowerMode(POWER_STANDBY) &&          // place in
       standby
          setControlMode(CTRL_INIT_START_FREQ) && // init start
             freq
          setControlMode(CTRL_START_FREQ_SWEEP))) // begin
             frequency sweep
          {
              return false;
          }

    // Perform the sweep. Make sure we don't exceed n.
    int i = 0;
    while ((readStatusRegister() & STATUS_SWEEP_DONE) !=
       STATUS_SWEEP_DONE) {
        // Make sure we aren't exceeding the bounds of our
           buffer
        if (i >= n) {
            return false;
        }

        // Get the data for this frequency point and store it
           in the array
        if (!getComplexData(&real[i], &imag[i])) {
            return false;
        }

        // Increment the frequency and our index.
        i++;
        setControlMode(CTRL_INCREMENT_FREQ);
    }

    // Put into standby
    return setPowerMode(POWER_STANDBY);
}

/**
 * Computes the gain factor and phase for each point in a frequency sweep.
 *
 * @param gain An array of appropriate size to hold the gain factors
 * @param phase An array of appropriate size to hold phase data.
 * @param ref The known reference resistance.
 * @param n Length of the array (or the number of discrete measurements)
 * @return Success or failure
 */
bool AD5933::calibrate(double gain[], int phase[], int ref, int n) {
    // We need arrays to hold the real and imaginary values temporarily
    int *real = new int[n];
    int *imag = new int[n];

    // Perform the frequency sweep
    if (!frequencySweep(real, imag, n)) {
        delete [] real;
        delete [] imag;
        return false;
    }

    // For each point in the sweep, calculate the gain factor and phase
    for (int i = 0; i < n; i++) {
        gain[i] = (double)(1.0/ref)/sqrt(pow(real[i], 2) + pow(imag[i], 2));
        // TODO: phase
    }

    delete [] real;
    delete [] imag;
    return true;
}

/**
 * Computes the gain factor and phase for each point in a
   frequency sweep.
 * Also provides the caller with the real and imaginary data.
 *
 * @param gain An array of appropriate size to hold the gain
   factors
 * @param phase An array of appropriate size to hold the phase
   data
 * @param real An array of appropriate size to hold the real
   data
 * @param imag An array of appropriate size to hold the
   imaginary data.
 * @param ref The known reference resistance.
 * @param n Length of the array (or the number of discrete
   measurements)
 * @return Success or failure
 */
bool AD5933::calibrate(double gain[], int phase[], int real[],
   int imag[],
                        int ref, int n) {
    // Perform the frequency sweep
    if (!frequencySweep(real, imag, n)) {
        return false;
    }

    // For each point in the sweep, calculate the gain factor
       and phase
    for (int i = 0; i < n; i++) {
        gain[i] = (double)(1.0/ref)/sqrt(pow(real[i], 2) + pow(
           imag[i], 2));

    }

    return true;
}
float AD5933::GetHumidity(){
```

```
int real[NUM_INCR + 1], imag[NUM_INCR + 1];
 double impedance[3][NUM_INCR + 1], mate[NUM_INCR + 1]={0}; double gain[NUM_INCR + 1]; double mate1=0; double avg=0; double X=0; int hum=0; double temp=0;
 double actImpedance[3][NUM_INCR + 1], phase[NUM_INCR + 1], imagimpedance[NUM_INCR + 1], capacitance[NUM_INCR + 1];
 for (int j = 0; j < 3; j++) {

   if (AD5933::frequencySweep(real, imag, NUM_INCR + 1)) {
     // Print the frequency data
     double cfreq = START_FREQ-100;
     cfreq += FREQ_INCR;
     for (int i = 0; i < NUM_INCR + 1; i++, cfreq += FREQ_INCR ) {
       double magnitude = sqrt(pow(real[i], 2) + pow(imag[i], 2));
       double gain[11] = {0.00000000050641708374,0.00000000051234250068,0.000000000

       impedance[j][i] = 1 / (magnitude * gain[i]);
       actImpedance[j][i]= (impedance[j][i]+371548)/5.65;


 double angle = atan (double(abs(imag[i]))/double(abs(real[i])));
 if(real[i]>0 && imag[i]>0){
   phase[i]= angle*M_PI;
 }
 if(real[i]<0 && imag[i]>0){
   phase[i]= 180-(angle*M_PI);
 }
 if(real[i]<0 && imag[i]<0){
   phase[i]= 180+(angle*M_PI);
 }
 if(real[i]>0 && imag[i]<0){
   phase[i]= 360+(angle*M_PI);
 }
 Serial.print("P: ");
 Serial.print(i + 1);
 Serial.print(": ");
 Serial.println(phase[i],20);

imagimpedance[i]= actImpedance[j][i]*sin(phase[i]);
capacitance[i]= 1/(2*M_PI*cfreq*imagimpedance[i]);
Serial.print("Capacitance:");
Serial.println(capacitance[i],20);
      }

    }

}

 for (int i=0; i<NUM_INCR + 1; i++){
    for (int j=0; j<3; j++){
      mate[i]+= actImpedance[j][i];
       //Serial.println(mate[i], 20);
    }
    Serial.println(mate[i], 20);
    mate[i] = mate[i] / 3;
    Serial.print(i + 1);
    Serial.print(":␣");

  }

  for(int i=0; i<NUM_INCR + 1; i++){
    mate1+= capacitance[i];
  }
  avg= mate1/(NUM_INCR + 1);


  X= avg/(1.8*pow(10,-10));
  hum = (int)(-3.4656*pow(10,3)*pow(X,3))+(1.0732*pow(10,4)*pow
     (X,2))-(1.0457*pow(10,4)*X)+(3.2459*pow(10,3));
  return hum;
}
```

C.1.3 Main Sketch

```
#include "LowPower.h"
#include <Wire.h>
#include <UnoWiFiDevEd.h>// if you are using the Arduino IDE
   1.8.x then: //#include <UnoWiFiDevEd.h>
#include "AD5933.h"
#include <Math.h>


#define CONNECTOR     "rest"
```

```
#define SERVER_ADDR   "api.thingspeak.com"

#define APIKEY_THINGSPEAK  "DVM9K6LGS8EKOCHL" //Insert your API
    Key

#define START_FREQ  (11300)
#define FREQ_INCR   (100)
#define NUM_INCR    (10)
#define REF_RESIST  (201200)

double gain[NUM_INCR + 1];
int phase[NUM_INCR + 1];
int real[NUM_INCR + 1];
int imag[NUM_INCR + 1];

int gasSensor = 15; // select input pin for gasSensor
int val = 0; // variable to store the value coming from the
   sensor
int Ro= 16000;
int sensePin = 14; // Pin A0
float sensorValue = 0; //Set the sensor values as a floating
   number
float kelvinValue = 0;
float celsiusValue = 0;
double hum = 0;
double temp1 = 0;
double temp2 = 0;
float gas1 = 0;
float gas2 = 0;
float gas3 = 0;
float gas4 = 0;
float gas5 = 0;


void setup() {
  Wire.begin();
  // Begin serial at 9600 baud for output
  //Serial.begin(9600);

  if (!(AD5933::reset() &&
        AD5933::setInternalClock(true) &&
        AD5933::setStartFrequency(START_FREQ) &&
        AD5933::setIncrementFrequency(FREQ_INCR) &&
        AD5933::setNumberIncrements(NUM_INCR) &&
        AD5933::setPGAGain(PGA_GAIN_X1)))
  {
    while (true) ;
  }

  // Perform calibration sweep
  AD5933::calibrate(gain, phase, REF_RESIST, NUM_INCR + 1);


  Ciao.begin(); // CIAO INIT
}

void loop() {

    field1();
    field2();
    field3();
    field4();
    field5();
    field6();
    field7();
    field8();
  for (int i = 0; i < 10; i++) {
  LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
     TIMER0_OFF,
                SPI_OFF, USART0_OFF, TWI_OFF);
  }
}

double celsius(){
  sensorValue = analogRead(sensePin); //reads voltage on Pin A0
  kelvinValue = (((sensorValue / 1023.0) * 5.0) * 100.0)-10;
  celsiusValue = kelvinValue - 273.15;
  return celsiusValue;
}

float Hydro() {

val = analogRead(gasSensor);// read the value from the pot
float voltage= val * (3.3 / 1023.0);
float Rs = ((5/voltage) - 1)*1000;
float ratio = Rs/Ro;
float y = -0.138*log(ratio)+0.6585;
return y;
}
float Ethan(){

val = analogRead(gasSensor);// read the value from the pot
float voltage= val * (3.3 / 1023.0);
float Rs = ((5/voltage) - 1)*1000;
float ratio = Rs/Ro;
float y = -0.11*log(ratio)+0.6097;
return y;
}

float Iso(){

val = analogRead(gasSensor);// read the value from the pot
float voltage= val * (3.3 / 1023.0);
float Rs = ((5/voltage) - 1)*1000;
float ratio = Rs/Ro;
float y = -0.11*log(ratio)+0.6521;
return y;
}

float Meth(){
val = analogRead(gasSensor);// read the value from the pot
float voltage= val * (3.3 / 1023.0);
float Rs = ((5/voltage) - 1)*1000;
float ratio = Rs/Ro;
float y = -0.063*log(ratio)+1.0202;
return y;
}

float Carbon(){

val = analogRead(gasSensor);// read the value from the pot
float voltage= val * (3.3 / 1023.0);
float Rs = ((5/voltage) - 1)*1000;
float ratio = Rs/Ro;
float y = -0.142*log(ratio)+1.0647;
return y;

}

```

```
void field1() {
  AD5933::GetHumidity();
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  hum = AD5933::GetHumidity();
  uri += "&field1=";
  uri += String(hum);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field2() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  temp1 = AD5933::getTemperature();
  uri += "&field2=";
  uri += String(temp1);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field3() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  temp2 = AD5933::getTemperature();
  uri += "&field3=";
  uri += String(temp2);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field4() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  gas1 = Hydro();
  uri += "&field4=";
  uri += String(gas1);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field5() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  gas2 = Ethan();
  uri += "&field5=";
  uri += String(gas2);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field6() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  gas3 = Iso();
  uri += "&field6=";
  uri += String(gas3);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field7() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  gas4 = Meth();
  uri += "&field7=";
  uri += String(gas4);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}

void field8() {
  String uri = "/update?api_key=";
  uri += APIKEY_THINGSPEAK;
  gas5 = Carbon();
  uri += "&field8=";
  uri += String(gas5);
  CiaoData data = Ciao.write(CONNECTOR, SERVER_ADDR, uri);
  delay(40000);
}
```

References

[1] UCTronics. (2015) Tgs2600 to-5 air gas sensor. [Online]. Available: http://www.uctronics.com/tgs2600-to-5-air-gas-sensor.html

[2] *1 MSPS, 12-Bit Impedance Converter, Network Analyzer*, Analog Devices, 2017, rev. F.

[3] R. C. P. Ltd. A000133—arduino uno wifi development board—arduino. [Online]. Available: http://au.rs-online.com/web/p/processor-microcontroller-development-kits/1113737/

[4] *3 V/5 V CMOS 0.5 SPDT/2:1 Mux in SC70*, Analog Devices, 2004, rev. 0.

[5] J. H.-F. G. P. Ltd. (2016) Cygnett chargeup digital 4000 portable power bank (green/grey). [Online]. Available: https://www.jbhifi.com.au/phones/all-phones/cygnett/cygnett-chargeup-digital-4000-portable-power-bank-green-grey/990236/

[6] *TGS 2600 - for the detection of Air Contaminants*, Figaro Engineering, Inc., 2013.

[7] I. AccuWeather. (2017) Sydney weather - accuweather forecast for new south wales australia. [Online]. Available: https://www.accuweather.com/en/au/sydney/22889/weather-forecast/22889

[8] G. Hunter, J. Stetter, P. Hesketh, and C. Liu, “Smart sensor systems,” *The Electrochemical Society*, pp. 29–34, 2010.

[9] B. F. Spencer, M. E. Ruiz-Sandoval, and N. Kurata, “Smart sensing technology: Opportunities and challenges,” *Structural Control and Health Monitoring*, pp. 349–368, Sep. 2004.

[10] M. E. E. Alahi, A. Nag, A. N. Manesh, S. Mukhopadhyay, and J. Roy, *A Simple Embedded Sensor: Excitation and Interfacing*. B. George, Ed. Springer International Pu, 2017, pp. 111–138.

[11] M. E. E. Alahi, L. Xie, A. I. Zia, S. Mukhopadhyay, and L. Burkitt, “A temperature compensated smart nitrate-sensor for agricultural industry,” *IEEE Transactions on Industrial Electronics*, pp. 1–8, 2017.

[12] S. D. T. Kelly, N. K. Suryadevara, and S. C. Mukhopadhyay, "Towards the implementation of iot for environmental condition monitoring in homes," *IEEE Sensors Journal*, vol. 13, no.10, pp. 3846–3853, Oct. 2013.

[13] D. Bhattacharjee and R. Bera, "Development of smart detachable wireless sensing system for environmental monitoring," *International Journal on Smart Sensing And Intelligent Systems*, vol. 7, no.3, pp. 1239–1253, Sep. 2014.

[14] A. Burns, B. R. Greene, T. J. McGrath, Michael J.and OShea, B. Kuris, S. M. Ayer, F. Stroiescu, and V. Cionca, "Shimmer a wireless sensor platform for noninvasive biomedical research," *IEEE Sensors Journal*, vol. 10, no.3, pp. 1527–1534, Sep. 2010.

[15] F. Sheikh and L. Xinrong, "Wireless sensor network system design using raspberry pi and arduino for environmental monitoring applications," in *The 9th International Conference on Future Networks and Communications (FNC-2014)*, Texas, USA, 2014, pp. 103–110.

[16] *HS1101LF Relative Humidity Sensor*, TE Connectivity Ltd., 9 2015.

[17] *LMx35, LMx35A Precision Temperature Sensors,LM335 datasheet*, Texas Instruments, 2 2015, rev. E.

[18] Arduino. (2017) Getting started with arduino and genuino products. [Online]. Available: https://www.arduino.cc/en/Guide/HomePage