**ADDIS ABABA INSTITUTE OF TECHNOLOGY**

**SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING**

SENIOR SEMESTER PROJECT PROGRESS REPORT

REALTIME WIRELESS WEATHER STATION (RTWWS)

Prepared by:

Mathias Assefa ---------------------- TCR/2295/02

Tsegaye Atsbeha --------------------- TCR/2690/02

Dagim Feyessa ------------------------- TCR/1892/02

Project Advisor:

Yalew Kidane

February 2014

Addis Ababa, ETHIOPIA

Table of Contents

[ACKNOWLEDGEMENT 2](#_Toc379777864)

[ABSTRACT 3](#_Toc379777865)

[INTRODUCTION 4](#_Toc379777866)

[Design 5](#_Toc379777867)

[High Level Design 6](#_Toc379777868)

[Component and Connector View 6](#_Toc379777869)

[Module View 7](#_Toc379777870)

[Detailed Design 8](#_Toc379777871)

[Central Control Station 8](#_Toc379777872)

[Data Collection Center 12](#_Toc379777873)

[Sensor Interface 16](#_Toc379777874)

[Accuracy of Reading 21](#_Toc379777875)

[Pressure Accuracy (MPX5050D): 21](#_Toc379777876)

[Temperature Accuracy (LM335): 21](#_Toc379777877)

[Light Accuracy (PhotoResesitor): 21](#_Toc379777878)

[DISCCUSSION OF RESULTS AND SUGGESTIONS FOR FURTHER WORK 22](#_Toc379777879)

[Results 22](#_Toc379777880)

[Temperature Data: 22](#_Toc379777881)

[Light Intensity Data: 22](#_Toc379777882)

[Pressure Data 23](#_Toc379777883)

[Suggestion for Further Work 23](#_Toc379777884)

[CONCLUSION 24](#_Toc379777885)

[REFERENCE 25](#_Toc379777886)

[Appendix A (MPX5050D) 26](#_Toc379777887)

[Appendix B (LM335) 28](#_Toc379777888)

[Appendix C (RabbitMQ Service) 29](#_Toc379777889)

[Appendix D (Arduino) 31](#_Toc379777890)

[Appendix E (Raspberry PI) 32](#_Toc379777891)

# ACKNOWLEDGEMENT

# ABSTRACT

Owing to the fact that weather forecast involves sophisticated equipment, high processing supercomputers and well trained manpower, makes it less available for people with low income and financial problems. Nowadays, with the advent of technology, weather data is available on several devices and various forms. Even though weather data is available on the internet, mobile devices and so on, it is not easy to get access to the real time weather data of a specific location. Real Time Wireless Weather Station, therefore, aims to solve this problem.

The project, being a two semester plan, we have been searching, and surveying for resources, methods and techniques that would help us to achieve our goal and make it reality. Our objective is to design, and prototype real time wireless weather station which provides complete weather data of a specific geographical location (for instance AAiT Campus’ Green Area). So that the user can get access to the data via web-browser or mobile app using internet.

# INTRODUCTION

These days, most of our lives depend not only on the past events and/or the future but also highly on the current ones. It is better understood getting enough data about the current situation would give an insight to the future. This is one of our initiative ideas behind the project choice and motivation that let us work on it. Our daily life is highly influenced by the weather condition surrounding us. Irrespective of what profession people have; farmer, teacher, politician, astronaut, leader, lawyer, and so forth wherever they live, there is an intersection point that makes us all curious about the weather around.

Of course we know that weather data is available, nowadays, more abundantly than ever. But is it sufficient? Do we have accurate and reliable weather data in hand? The answer is obvious, for most of the society in the world, it is not. At least for our country, Ethiopia, not even close. Have you ever wondered when you saw the weather forecast for Addis Ababa that indicates it won’t be raining, and when you go out for shopping you got yourself in trouble? Perhaps, that would not have happened, if you had access to weather data (forecast) of a specific location in real time. And this would make it more reliable if more locations have such weather stations that gather and send data to a central system whereby anyone can simply browse and see what is going on around.

# Design

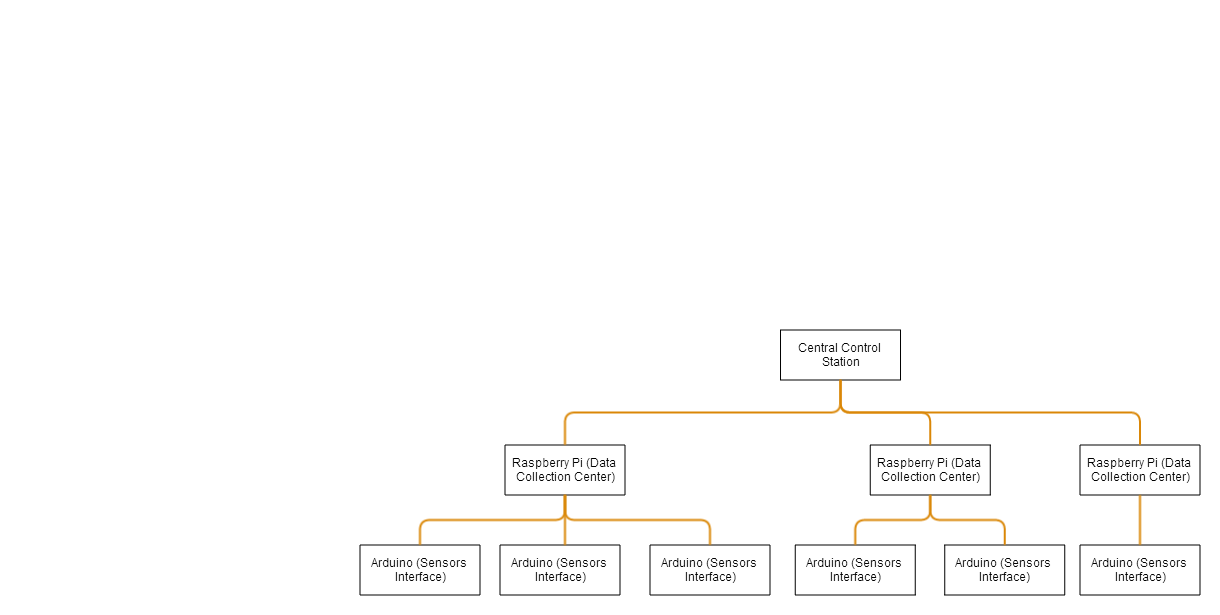
Basically, data is collected with sensors (temperature, pressure, humidity, light intensity, rain, wind etc…). Analog data is collected by the various sensors incorporated which is then processed by hardware (in this case Arduino UNO is used). The analog data is digitized by the Arduino (ADC) and then forwarded to the data collection center (Raspberry Pi). The raspberry pi provides wireless communication with the central control station whereby data is processed for further use and maintenance. The technical details of these system will be discussed in detail later on.

Therefore, from the hardware side the basic functions are raw data collection (analog data), analog to digital conversion (digitizing data), and then send data to central server (wireless communication). From the software side, programs are written to control these activities, process data, and user interface whereby user can see weather data. There should be system management module whereby administrator (authorized personnel) make changes and/or send configuration commands to the weather station(s).

## 

## High Level Design

### Component and Connector View



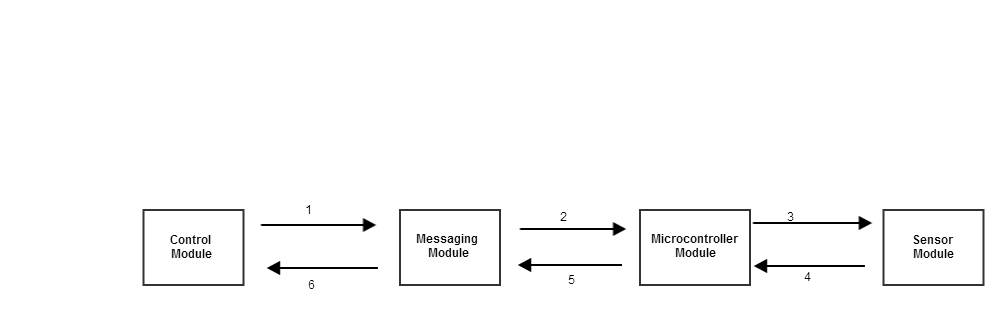
1. The Central Station

* The central control station is a server that will later be used to implement the weather prediction service. It is where all the data collected from the various data centers are sent for further analysis. It also contains the messaging service (Rabbit MQ Service (Appendix A))

1. The Data Collection Center
   * The data collection center, is a raspberry pie (Appendix A) that collects the weather conditions from several Arduino and sends them to the central station.
   * The main purpose of implementing a data collection center is to extend the area we can collect the weather from. Since the raspberry pi has internet access it allows us to extend the range we can send the data. It’s because of this range that we are able to collect the data from a very large area but still have one control station of having multiple ones.
   * The data collection center is designed in a way it is small and requires little to no maintenance.
2. The Sensor interface

* It is where all the sensors are located, in the current version only a temperature, light and pressure sensor are used.
* It’s implemented by the use of an Arduino.
* The Arduino is used to get the data from the various sensors connected to it then, it transmits it to the data collection.
* In the current version we are using serial communication to transfer data and commands to and from the data collection center.

### Module View



1. The control module sends command (using RabbitMQ) to the messaging module (for a specific raspberry pi and Arduino/microcontroller) whenever it requires a certain data. Such as device status, and readings.
2. The messaging broker receives commands from the control module which in turn transfer the command to a specific microcontroller. It main function is for routing messages and/or commands using RabbitMQ services.
3. The microcontroller module reads the state/value of the sensor and executes command based on the command received from the messaging module. This is done through serial communication.
4. Sensor module is used to gather weather conditions. Based on the states set by the microcontroller, sensors can be set to state ON or state OFF. Which later on the data is read and processed by the microcontroller.
5. The microcontroller, reads and process data from the sensor module. Since analog data is read by the sensors, it should be digitized for further process and usage (forwarded to the messaging module).
6. Finally, the messaging module receives processed data. These data is then displayed to the end user.

## Detailed Design

### Central Control Station

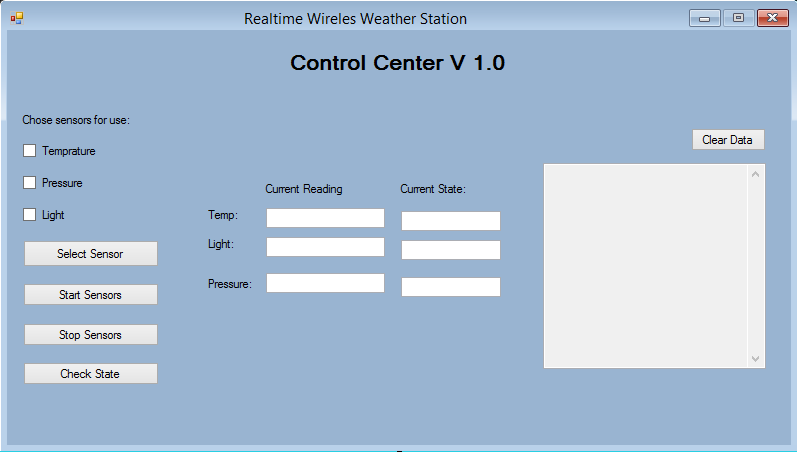


Fig 1. Central control Station

In the current version since only one device is currently in use, the central station does not provide a choice of device, but the underlying system is designed and implemented in a way to support multiple devices (as shown in the component and connectors view).

The above figure shows the user interface to control a specific device. In later version of this project there will be a previous window which will allow the user to choose which device to control from a list of them.

The central control station has two important task it requires to achieve (as shown in the module view design). These Two tasks are:

1. Routing to a specific device

Routing to a specific device is achieved by the use of topic based message exchange that is provided by the RabbitMQ Service. The reason for using RabbitMQ services for messaging are, due to the fact it is machine independent, open source, commercially supported, and has flexible routing. Topic based exchange is a method in which every message has an attached routing key, this routing key can be composed of several parameter delimited by a period (‘.’).

The routing key that is used in this project is:

<Message Recipient type>.<Data Center Name>.<Sensor Interface Name>

Example 1: Device.RaspB1.Ardu4 this means that the message is sent to the devices and the device is one in which the data center is RaspB1, and the sensor interface is Ardu4.

Example 2: Control.RaspB1.Ardu4 this means that the message is sent to the central control station and the device that sent this message is one in which the data center is RaspB1, and the sensor interface is Ardu4.

The RabbitMQ service also has two special character (‘#’ ‘\*’) which further extend its routing capabilities.

The # function is used to receive all the messages, regardless of the routing key, but it is more powerful when used in conjecture with a routing key

For Example Control.# will receive all message to Control regardless of the number of parameters and type of parameter.

The \* function is used to receive all the messages, regardless of the routing parameter, but it is more powerful when used in conjecture with a routing key

For Example Device.\*.\* will send message to all the devices regardless type of parameter. This is especially useful if it is necessary to send a system wide command.

Sending Command Using RabbitMQ:

In order to send message using the RabbitMQ service, the RabbitMQ C# client library was used and the RabbitMQ.Client library was included. The method that was used in the project in order to send message from the Central Control Station is:

public void sendData(string message, string routingTemp, string server, string exchange)

{

//chose the ipaddress for the RabbitMQ service

var factory = new ConnectionFactory() { HostName = server };

//create a connection to the service

using (var connection = factory.CreateConnection())

{

//open a channel of communication with the service

using (var channel = connection.CreateModel())

{

//set the place of message exchange, and the type of exchange to topic

channel.ExchangeDeclare(exchange, "topic");

//set the routing key for the message

var routingKey = routingTemp;

// change the encoding of the message to make it safer to transfer

var body = Encoding.UTF8.GetBytes(message);

//publish the message to the specified exchange with the specified routing key

channel.BasicPublish("topic\_logs", routingKey, null, body);

}

}

}

As can be seen the method allows for easy change of the routing key the exchange and even the server, so it is quite easy to improve on the scale of the current version of the project since the routing framework is already pre-built.

Receiving Data from Data Centers:

In order to send message using the RabbitMQ service, the RabbitMQ C# client library was used. The RabbitMQ.Client library, and the RabbitMQ.Client.Events library were included. The method is run in a separate thread, which raises an event any time a message is received. The method that was used in the project in order to receive message from the Data Center is:

private void receiveMessage()

{

// create a connection between the RabbitMQ service located at the given location

var factory = new ConnectionFactory() { HostName = serverAddress };

using (var connection = factory.CreateConnection())

{

using (var channel = connection.CreateModel())

{

//set the exchange name and type to listen to

channel.ExchangeDeclare("topic\_logs", "topic");

//create a temporary and random que to store the message received from the exchange

var queueName = channel.QueueDeclare();

// bind the exchange and routing key to the random que created

channel.QueueBind(queueName, "topic\_logs", "Control.\*.\*");

Console.WriteLine(" [\*] Waiting for messages. " +

"To exit press CTRL+C");

// create a consumer which can read the messages from the que

var consumer = new QueueingBasicConsumer(channel);

//assign the consumer to the random que

channel.BasicConsume(queueName, true, consumer);

// continuously loop and receive message from the exchange

while (true)

{

//created whenever a message is received

var ea = (BasicDeliverEventArgs)consumer.Queue.Dequeue();

// retrieve the message that was sent

var body = ea.Body;

// Decode the message

var message = Encoding.UTF8.GetString(body);

//retrieve the routing key of the sender

var routingKey = ea.RoutingKey;

Console.WriteLine(" [x] Received '{0}':'{1}'",

routingKey, message);

//store the received data

msg.Message = routingKey + ":" + message;

}

}

}

As can be seen from the code above it is easy to change the routing key, exchange and the server. Also because the routing key that is being used is the Control.\*.\* it allows us to receive message from all the devices using just this method.

1. Sending commands to a specific Device based on user choice

The control station has 5 commands it sends to a device, this commands are:

* Start sensing: When the start sensor button is clicked, then a command is sent to the specified sensor to turn all its sensors ON.
* Stop sensing: When the stop sensor button is clicked, then a command is sent to the specified sensor to turn all its sensors OFF.
* Select sensors: When the select sensor button is clicked, then a command is sent to the specified sensor to turn its sensors on and off based on the checked box for each.
* Check Sensor State: Checks to see which sensors are ON and which are OFF. The returned data from this device is displayed in the current state column of the sensors.
* Read Data: This is a command that is sent from the central control station whenever we want a reading of sensor values. The system is currently implemented in a way that we send this command in a separate thread ever few second.

1. Receiving data from all devices and Displaying it

The control station has two main data it receives from the devices:

* Sensor State: Indicate which sensors are turned ON and which sensor are turned OFF.
* Sensor Value: Indicate what the current readings of the sensors are.

Both of the above reading are received using the method shown in routing, but are parsed to retrieve the necessary information from them.

### Data Collection Center

In the current version of this project it is implemented on a Raspberry Pi (Appendix B). Its main purpose is to receive command from the Central Command Station and communicate this command to the Sensor interface for execution. In the current version of the project, the communication means that is being used is serial communication. The reason we have chosen serial communication is due to the fact that it was the most readily available means of communication that was present, but for future version we want it to be wireless communication.

The Data collection Center is written in the python langue due to the fact that the Raspberry pi uses a Linux OS, and python is a light weigh language which is a key factor considering the limited power for the raspberry pi. The Data collection Center has 2 significant task it has to accomplish:

1. Receive Command from Central Command Station

This is the start of any task. It is constantly running on the device, until a message is received from the control station, when the message is received it sends this message to the sensor interface (Arduino) using serial communication. The library that was used to allow RabbitMQ service manipulation in python is Pika, and the library used for serial communication is pyserial. No actual processing is done on the message in the current version, but for future versions message caching and selective device messaging will be implemented on the Data collection center.

1. Send Data to Central Command Station

This task is done right after task 1 is completed. As soon as the command that was received is sent to the sensor interface, the data collection center start to wait to receive data from the sensor interface using serial communication. Then it sends this data back to the control station using a routing key which looks like control.Raspberryname.Arduino name. When this is done then it goes back to task 1 (waiting to receive command from the central station).

The code that achieves the above task is:

#import the necessary librarys

import pika

import sys

import serial

#create a connection with the specified ipaddress (localHost)

connection = pika.BlockingConnection(pika.ConnectionParameters( host='localhost'))

#create a channel to connect with the RabbitMQ service

channel = connection.channel()

#create a global variable containing for serial communication

global ser

#set the serial communication to use COM30 and at a speed of 9600B baud rate

ser = serial.Serial('COM30', 9600)

#variable indicating if there is a message to receive from the serial port

global messageReceived

messageReceived = False

#chose the exchange to listen to (topic\_logs)

channel.exchange\_declare(exchange='topic\_logs', type='topic')

#create a random que to temporarily store the message

result = channel.queue\_declare(exclusive=True)

queue\_name = result.method.queue

#obtain the binding key (routing key) for the que from the system argument

#it should be the Raspberry name. The Arduino name

binding\_keys = "Device."+sys.argv[1]

#check to see if binding key is provided with system argument

if not binding\_keys:

print >> sys.stderr, "Usage: %s [binding\_key]..." % (sys.argv[0],)

sys.exit(1)

#if binding key is provided bind the key to the random que created

for binding\_key in binding\_keys:

channel.queue\_bind(exchange='topic\_logs', queue=queue\_name, routing\_key=binding\_key)

print ' [\*] Waiting for logs. To exit press CTRL+C'

#method that is called whenever a message is received

def callback(ch, method, properties, body):

print " [x] %r:%r" % (method.routing\_key, body,)

#send the command that was received directly to the serial port

ser.write(body+'\n')

#set the message received status true

messageReceived = True

#as long as there is a message to be received do

while messageReceived:

#try to read a serial communication from the serial port

try:

message = ser.readline()

#allow for keyboard interrupt to stop system from getting stuck

except KeyboardInterrupt :

#if interrupt occurs close the serial communication

ser.close()

break

#set the routing key for sending message

routing\_key = "Control."+sys.argv[1]

#publish the message to topic\_logs with the given routing key, and message read from the serial port

channel.basic\_publish(exchange='topic\_logs',

routing\_key=routing\_key,

body=message)

print " [x] Sent %r:%r" % (routing\_key, message)

# stop trying to receive message from the serial port

messageReceived = False

# Receive the message from the random que, calls the callbak method

channel.basic\_consume(callback,

queue=queue\_name,

no\_ack=True)

# starts the recive message method

channel.start\_consuming()

### 

### Sensor Interface

Is implemented using the Arduino framework. The reason for the choice of the Arduino is the availability of the material and the ease of use for prototyping purpose. The main purpose of the Arduino is to create a bridge between the analogue signals received from the sensors and the digital signals required by the Data collection center. There are two main task that the sensor interface does:

1. Read Serial Communication

This is done by the use of the inbuilt serial event (interrupt). After every execution of the main loop, the serial port is checked to see if there is a message to be received, if a message exists then the serial event method is run. The serial event is mainly concerned with parsing the command received into meaningful and useful sections. Generally the message is parsed into the command section and the parameter used by in the execution of the commands.

The code for this is given below:

void serialEvent()

{

// used to make sure the code is only executed when there is

// more serial communication to be read

while (Serial.available())

{

// if its the first char to be read store it separatly as a command character

if (stringNum==0)

{

cmdChar = (char)Serial.read();

//increment the number of read character by one

stringNum++;

}

// if its the secound character and if the command is to change state

// then read the secound char and store it as inout state to be

// used later for setting the state of the sensors

if (cmdChar==changeState&&stringNum==1)

{

inputState = (char)Serial.read();

}

// if its the secound char and the comand is not change state or if its the

// third char store the info into a string (just a precaution not used for the moment)

if ((stringNum>1&&cmdChar!=changeState)||stringNum>2)

{

inChar = (char)Serial.read();

// add the reead char to the inputString:

inputString += inChar;

}

//Serial.println(inputString);

// if the incoming character is a newline, set a flag

// so the main loop can do something about it: also set some variables to default values for reuse

stringNum++;

if (inChar == '\n')

{

stringNum=0;

stringComplete = true;

inChar=0;

}

}

}

1. Send Serial Communication

This is done by the main loop. It executes all the command received, and resets the various variables for reuse. And then depending on the command received it send back a reply. This is the part in which the sensor status is read, set, the sensor reading are read.

The code written to achieve this is given below:

void loop()

{

//if we have completed reading all the message that was sent

//and the command is to change state then do the following

if (stringComplete && cmdChar==changeState)

{

// send the comand that was provided back

Serial.print(cmdChar);

Serial.print('|');

//method that takes in the read value and based on that change state of the sensor

//input state is basically the binary rep of the state in dec format

changeSensorState(inputState);

// send 1 confirming sucess

Serial.println('1');

// clear the values that are used for reuse

inputString = "";

stringComplete = false;

cmdChar = 0;

}

//if reading the message is done and if the command is to

// check sensor state then do

if (stringComplete && cmdChar==checkSensor)

{

// send what kind of data is separated by '|' from the actual message

Serial.print(cmdChar);

Serial.print('|');

// method that outputs a string of the state of the sensors

// in the order of temp, light and pressure

Serial.println(checkState());

// clear the values that are used for reuse

inputString = "";

stringComplete = false;

cmdChar = 0;

}

//if reading the message is done and if the command is to

// stop Sensing

if (stringComplete && cmdChar==stopSensing)

{

// send the comand that was provided back

Serial.print(cmdChar);

Serial.print('|');

//change the sensors state of all to low

changeSensorState('0');

// send 1 confirming sucess

Serial.println('1');

// clear the values that are used for reuse

inputString = "";

stringComplete = false;

cmdChar = 0;

}

//if reading the message is done and if the command is to

// start sensing

if (stringComplete && cmdChar==startSensing)

{

// send the comand that was provided back

Serial.print(cmdChar);

Serial.print('|');

//change the sensors state of all to high

changeSensorState('7');

// send 1 confirming sucess

Serial.println('1');

// clear the values that are used for reuse

inputString = "";

stringComplete = false;

cmdChar = 0;

}

//if reading the message is done and if the command is to

// read sensed values

if (stringComplete && cmdChar==readValue)

{

// send what kind of data is separated by '|' from the actual message

Serial.print(cmdChar);

Serial.print('|');

//method used to read sensor data

readSensorValue();

// clear the values that are used for reuse

inputString = "";

stringComplete = false;

cmdChar = 0;

}

//a delay of half a secound inorder to allow the message api

// writtin in python to execute its tasks and start reading from the arduino(sensors)

delay(500);

}

# Accuracy of Reading

The accuracy of the readings is mainly dependent on the sensors used.

## Pressure Accuracy (MPX5050D):

* 2.5% Maximum Error over 0°to 85°C
* Temperature Compensated Over -40°to +125°C

## Temperature Accuracy (LM335):

* +10 mV/˚K
* calibrated at 25˚C the LM135 has typically less than 1˚C error over a 100˚C temperature range
* has a linear output

## Light Accuracy (PhotoResesitor):

* A range of 0.73V – 5V is directly converted into percentage
* The range is calibrated in order to get good sensitivity while maintaining good threshold
* Obtained through trial and error.

# DISCCUSSION OF RESULTS AND SUGGESTIONS FOR FURTHER WORK

## Results

According to the project timeline, we believe that the required goal has been achieved. The results and outcomes we found from the testing and experimentation are discussed below.

## Temperature Data:

Analog temperature data has been received from the temperature sensor and were able to log and display data on terminal window from Arduino. The received data was promising and precise. Sample readings are shown below.

## Light Intensity Data:

Analog light intensity data has been received from the light sensor and were able to log and display data on terminal window from Arduino. The received data was promising and precise. Sample readings are shown below.

## Pressure Data

Analog pressure data has been received from the pressure sensor and were able to log and display data on terminal window from Arduino. The received data was promising and precise. Sample readings are shown below.

# Suggestion for Further Work

For the first term of our project, we limited ourselves to do the complete system design; including high level and detail design. And we believe that our project can succeed if the necessary resources and materials are accessible. We understand, however, that our capacity is limited and all we can do is use every resource we have on hand use it effectively. Therefore, we believe prototyping with few sensors that can give an overview on the general weather station system and its working principles.

This being said, we know we have much more to be done by the next term. The weather station has to be complete. Here, the term complete means, it should function as it is needed as of the plan we designed in the beginning. In addition to receiving real time weather data, and displaying to the user, prediction mechanisms, geographical location, web-based user interface, mobile app and additional sensors should be incorporated.

# CONCLUSION

# REFERENCE

# Appendix A (MPX5050D)

Manufactured by: Freescale Semiconductor

The MPXx5050 series piezoresistive transducer is a state-of-the-art monolithic silicon pressure sensor designed for a wide range of applications, but particularly those employing a microcontroller or microprocessor with A/D inputs. This patented, single element transducer combines advanced micromachining techniques, thin-film metallization, and bipolar processing to provide an accurate, high level analog output signal that is proportional to the applied pressure.

Features

• 2.5% Maximum Error over 0°to 85°C

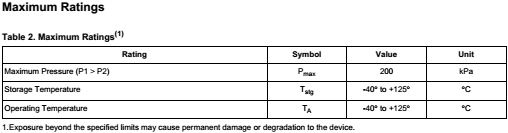
• Ideally suited for Microprocessor or Microcontroller-Based Systems

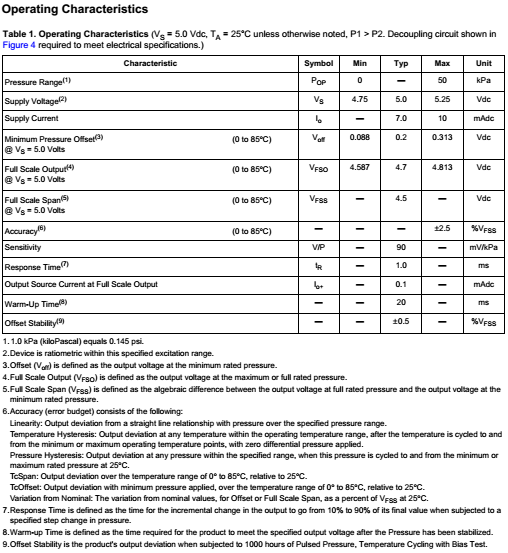
• Temperature Compensated Over -40°to +125°C

• Patented Silicon Shear Stress Strain Gauge

• Durable Epoxy Unibody Element

• Easy-to-Use Chip Carrier Option





# Appendix B (LM335)

LM135/LM235/LM335, LM135A/LM235A/LM335A Precision Temperature Sensors

General Description

The LM135 series are precision, easily-calibrated, integrated circuit temperature sensors. Operating as a 2-terminal zener, the LM135 has a breakdown voltage directly proportional to absolute temperature at +10 mV/˚K. With less than 1Ω dynamic impedance the device operates over a current range of 400 µA to 5 mA with virtually no change in performance. When calibrated at 25˚C the LM135 has typically less than 1˚C error over a 100˚C temperature range.

Unlike other sensors the LM135 has a linear output. Applications for the LM135 include almost any type of temperature sensing over a −55˚C to +150˚C temperature range. The low impedance and linear output make interfacing to readout or control circuitry especially easy. The LM135 operates over a −55˚C to +150˚C temperature range while the LM235 operates over a −40˚C to +125˚C temperature range. The LM335 operates from −40˚C to +100˚C. The LM135/LM235/LM335 are available packaged in hermetic TO-46 transistor packages while the LM335 is also available in plastic TO-92 packages.

Features

* Directly calibrated in ˚Kelvin 1˚C initial accuracy available
* Operates from 400 µA to 5 mA
* Less than 1Ω dynamic impedance
* Easily calibrated
* Wide operating temperature range
* Low cost

# Appendix C (RabbitMQ Service)

Messaging enables software applications to connect and scale. Applications can connect to each other, as components of a larger application, or to user devices and data. Messaging is asynchronous, decoupling applications by separating sending and receiving data.

You may be thinking of data delivery, non-blocking operations or push notifications. Or you want to use publish / subscribe, asynchronous processing, or work queues. All these are patterns, and they form part of messaging.

RabbitMQ is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

Feature Highlights

* Reliability

RabbitMQ offers a variety of features to let you trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability.

* Flexible Routing

Messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic. For more complex routing you can bind exchanges together or even write your own exchange type as a plugin.

* Clustering

Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker.

* Federation

For servers that need to be more loosely and unreliably connected than clustering allows, RabbitMQ offers a federation model.

* Highly Available Queues

Queues can be mirrored across several machines in a cluster, ensuring that even in the event of hardware failure your messages are safe.

* Multi-protocol

RabbitMQ supports messaging over a variety of messaging protocols.

* Many Clients

There are RabbitMQ clients for almost any language you can think of.

* Management UI

RabbitMQ ships with an easy-to use management UI that allows you to monitor and control every aspect of your message broker.

* Tracing

If your messaging system is misbehaving, RabbitMQ offers tracing support to let you find out what's going on.

* Plugin System

RabbitMQ ships with a variety of plugins extending it in different ways, and you can also write your own.

# Appendix D (Arduino)

Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board.

Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. Arduino projects can be stand-alone, or they can communicate with software running on your computer (e.g. Flash, Processing, MaxMSP.) The boards can be assembled by hand or purchased preassembled; the open-source IDE can be downloaded for free.

The Arduino programming language is an implementation of Wiring, a similar physical computing platform, which is based on the Processing multimedia programming environment.

There are many other microcontrollers and microcontroller platforms available for physical computing. Parallax Basic Stamp, Netmedia's BX-24, Phidgets, MIT's Handyboard, and many others offer similar functionality. All of these tools take the messy details of microcontroller programming and wrap it up in an easy-to-use package. Arduino also simplifies the process of working with microcontrollers, but it offers some advantage for teachers, students, and interested amateurs over other systems:

* Inexpensive - Arduino boards are relatively inexpensive compared to other microcontroller platforms. The least expensive version of the Arduino module can be assembled by hand, and even the pre-assembled Arduino modules cost less than $50
* Cross-platform - The Arduino software runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows.
* Simple, clear programming environment - The Arduino programming environment is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with the look and feel of Arduino
* Open source and extensible software- The Arduino software is published as open source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.
* Open source and extensible hardware - The Arduino is based on Atmel's ATMEGA8 and ATMEGA168 microcontrollers. The plans for the modules are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.

# Appendix E (Raspberry PI Raspberry Pi Logo)

Earlier this year a small single board computer stamped with chips and I/O connectors — with the tasty name of Raspberry Pi — began receiving a great deal of press coverage. It has captured the curiosity of tech journalists and enthusiasts around the world due to a combination of its purpose, capabilities, and usefulness.

The Raspberry Pi is a small, barebones computer developed by The Raspberry Pi Foundation, a UK charity, with the intention of providing low-cost computers and free software to students. Their ultimate goal is to foster computer science education and they hope that this small, affordable computer will be a tool that enables that.

The printed circuit board (PCB) houses the input and output connectors as well as the computer hardware itself. Currently, the Foundation is selling a naked PCB — meaning there is no included Raspberry Pi case — and will release a cheaper version, with fewer connectivity options, soon. These two versions without cases are essentially the beta run and buildup to the release of the final product.  The final version will be an educational edition with case, documentation, and pre-loaded educational software. On the software side of things, there are currently three Linux-based operating systems supported by the Raspberry Pi.

There were 10,000 Model B versions produced in the first production batch, and they all sold out within hours of going on sale. The Model B is the $35 version with the most connectivity options, and is the model that most enthusiasts are interested in. A Model A without ethernet and with a single USB port is on the way and will be sold for $25. Furthermore, the final educational edition Raspberry Pi computers (with case and extras) are slated for a summer 2012 release, though the price is unknown.

**Specifications and performance**

As for the specifications, the Raspberry Pi is a credit card-sized computer powered by the Broadcom BCM2835 system-on-a-chip (SoC). This SoC includes a 32-bit ARM1176JZFS processor, clocked at 700MHz, and a Videocore IV GPU. It also has 256MB of RAM in a POP package above the SoC. The Raspberry Pi is powered by a 5V micro USB AC charger or at least 4 AA batteries (with a bit of hacking).

While the ARM CPU delivers real-world performance similar to that of a 300MHz Pentium 2, the Broadcom GPU is a very capable graphics core capable of hardware decoding several high definition video formats. However, in order to keep costs of the Raspberry Pi low, the UK charity has only licensed the H.264 codec for hardware decoding (and it is unclear if users will be able to purchase/activate additional codecs). In that regard, the Videocore IV GPU is rather potent as it is capable of hardware decoding 1080p30 H.264 with bit-rates up to 40Mb/s.

The Raspberry Pi model available for purchase at the time of writing — the Model B — features HDMI and composite video outputs, two USB 2.0 ports, a 10/100 Ethernet port, SD card slot, GPIO (General Purpose I/O Expansion Board) connector, and analog audio output (3.5mm headphone jack). The less expensive Model A strips out the Ethernet port and one of the USB ports but otherwise has the same hardware. It is this model that is the “$25 PC” that originally made so many headlines.