REMOTE WATER PUMP MONITORING SYSTEM

By

Masaki Sato
Raynaldi Yose Iskandar
Yun Mo Kang

Final Report for ECE 445, Senior Design, Fall Spring 2020
TA: Dean Biskup

09 December 2020
Project No. 18

Abstract

Having access to a clean source of water still remains a challenge in some parts of the world. Building water pumps in these areas is one part of the solution, while maintaining them with an effective method of monitorization is another. For such rural areas, a wireless remote water pump monitoring system would be an ideal solution. The monitor should be able to measure important values such as temperature and vibration and send those data to a server. This data should then be translated to a graphic representation on a website, so that anyone could see it and check if there is any problem on the water pump.

Contents

1. Introduction 1
   1.1 Problem 1
   1.2 Solution 1
   1.3 High-Level Requirements 1
2 Design 2
   2.1 Physical Design 2
   2.2 Block Diagram 3
   2.3 Hardware Design 4
      2.3.1 Power Module 4
      2.3.2 Sensor Module 5
   2.4 Software Design 7
      2.4.1 Sensor Data reading on EPS 32 7
      2.4.2 WiFi and AWS IoT communications on EPS 32 8
      2.4.3 AWS Lambda 8
      2.4.4 AWS Athena Database 9
      2.4.5 AWS S3 and Website Hosting 9
3. Design Verification 10
   3.1 Hardware Design Verification 10
      3.1.1 Power Module 10
      3.1.2 Sensor Module 11
   3.2 Software Design Verification 13
      3.2.1 Sensor Data reading on EPS 32 13
      3.2.2 WiFi and AWS IoT communications on EPS 32 14
      3.2.3 AWS Lambda 14
      3.2.4 AWS Athena 15
      3.2.5 AWS S3 and Website Hosting 15
4. Costs 17
   4.1 Parts 17
   4.2 Labor 17
   4.3 Schedule 18
5. Conclusion 19
   5.1 Accomplishments 19

5.2 Uncertainties 19
5.3 Ethical considerations 19
5.3.1 Ethics 19
5.3.2 Safety 20
5.4 Future work 20
References 21
Appendix A: Requirements and Verifications Table 22
Appendix B: ESP 32 C Code 27
Appendix C: AWS Lambda Python Code 30

1. Introduction

1.1 Problem

In remote regions of Indonesia, many rural villages are located very far from clean water sources. Out of 264 million people in Indonesia, 28 million lack safe water and 71 million lack access to improved sanitation systems [1]. Each day, villagers of a specific village named Nibaaf take three-hour trips [2] to get clean water. In order to fight this problem, a non-profit organization called *Solar Chapter*, affiliated by one of our team members, has been building water pump systems that deliver clean water to villages in vicinity. However, they lacked a means to monitor and maintain the water pumps. Any sort of downtime would have adverse effects on the villagers' well-being, therefore constant maintenance is crucial. However, due to the remoteness of the location, having regular inspection is troublesome.

1.2 Solution

Our team is proposing a solution in the form of a remote water pump monitoring system. The system takes the pump's basic operating data such as water flow and up-time measurement to monitor the pump's behavioral trend. The device would also regularly measure safety parameters including vibration and temperature of the water pump. The system would send an alert when it receives undesired values so the operator can send in maintenance. Performing these precautions can extend the longevity of these water pumps and prevent them from breaking down unexpectedly. This would also prevent any downtime and greatly improve the sustainability of the water system. The values measured from the sensors would then be transmitted remotely through a cellular network to a cloud-based database system, which then will be visualized through a website or an app. This will allow high accessibility for the operator. This system minimizes the need of physical onsite personnel presence to only emergency maintenance and longer-term physical inspections, while still keeping the water system dependable.

1.3 High-Level Requirements

- Microcontroller must process the output signal of each sensor of varying form and translate them to a quantitative value. These values include flow, current, temperature, and vibration.
- Collected data must be transferred successfully without corruption through cellular network connection to the cloud based database in an interval of 15 minutes.
- Once the cloud database is updated with new input from the microcontroller, the updated information should also be included in the spreadsheets on the database server. The website should reflect the new information every 24 hours.

2 Design

2.1 Physical Design

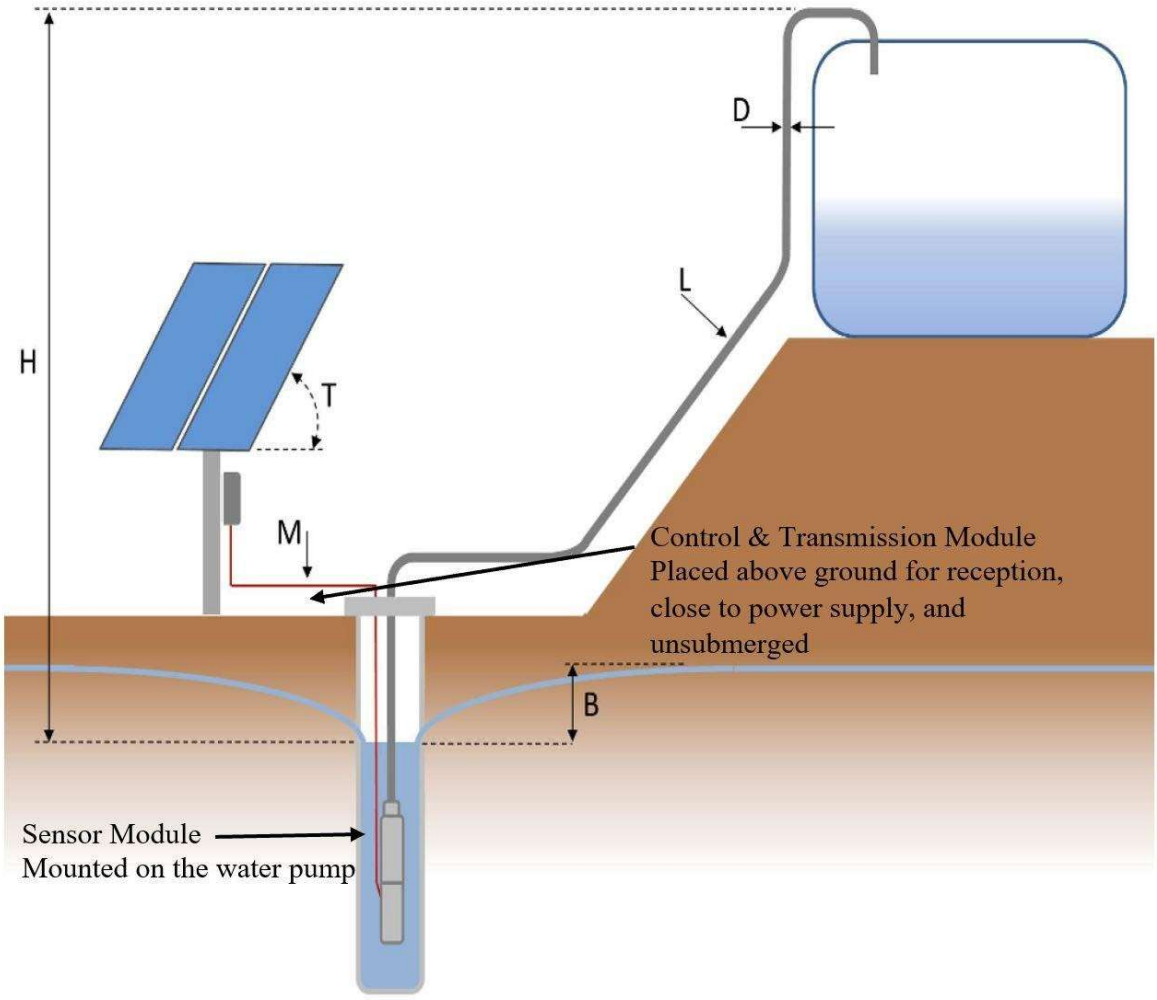

Figure 1: Placement of physical design on water pump system [3]

Figure 1 represents the real-life implementation of the system, with sensors mounted on the pump to monitor its parameters underwater, while power and control modules remain unsubmerged in the vicinity of the water source opening. For the prototype we have built, we did not concern ourselves with a waterproof design, as we believe that specification is rather trivial to implement. The real-life application also requires a rather long wiring between sensor module and control module, another aspect we ignored in the prototype. The monitoring system we designed serves as an add-on to an already existing water supply system. The pump itself is solar powered, hence the idea of implementing a self-sustaining monitoring system in the real-life application. Clean water is extracted from the source to be pumped to a reservoir, where further distribution happens to the villages nearby. Thus, several of these monitoring systems need to be deployed on the various pump sites.

2.2 Block Diagram

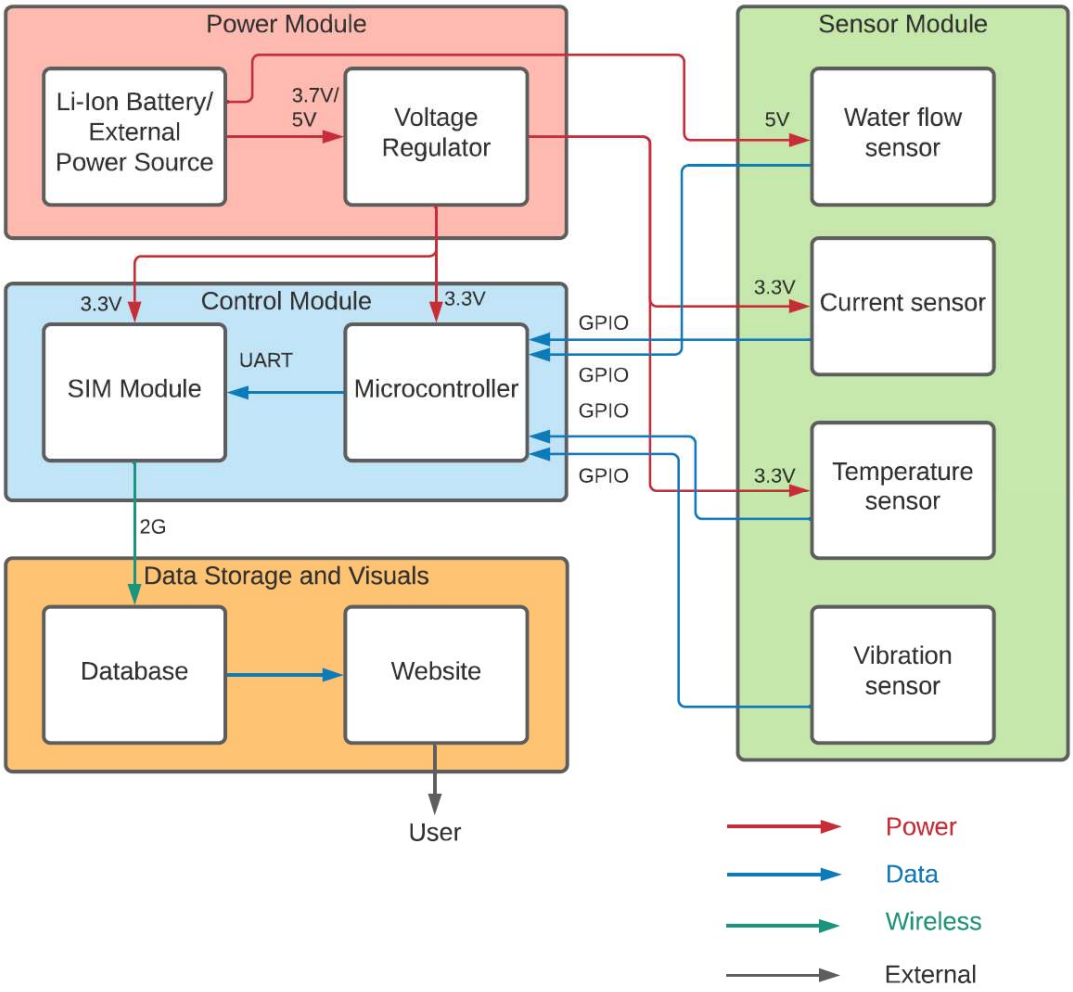

Figure 2: Block diagram of the remote monitoring system

The system's block diagram is shown in Figure 2. Firstly, the power module serves as the power supplier of the system. It mainly features a voltage regulator which regulates an arbitrary voltage source down to a 3.3 V power signal to act as the main power line of the system. Next is the sensor module, which can be categorized into two types: operational parameter sensor and safety parameter sensor. Operational parameter sensor consists of water flow sensor and current sensor, and is meant to monitor the pump's basic operational data; while safety parameter sensor consists of temperature sensor and vibration sensor, and is meant to give a warning flag once either one of the parameters start to go out of the norm, indicating a physical issue with the pump. Data from each sensor is sent to the control module where a microcontroller will process them to a quantifiable value, and then packed into a JSON format to be transmitted through the internet to a database in the data storage and visuals subsystem. The website will then read data from the database and visualize them through graphs for the user to view.

2.3 Hardware Design

2.3.1 Power Module

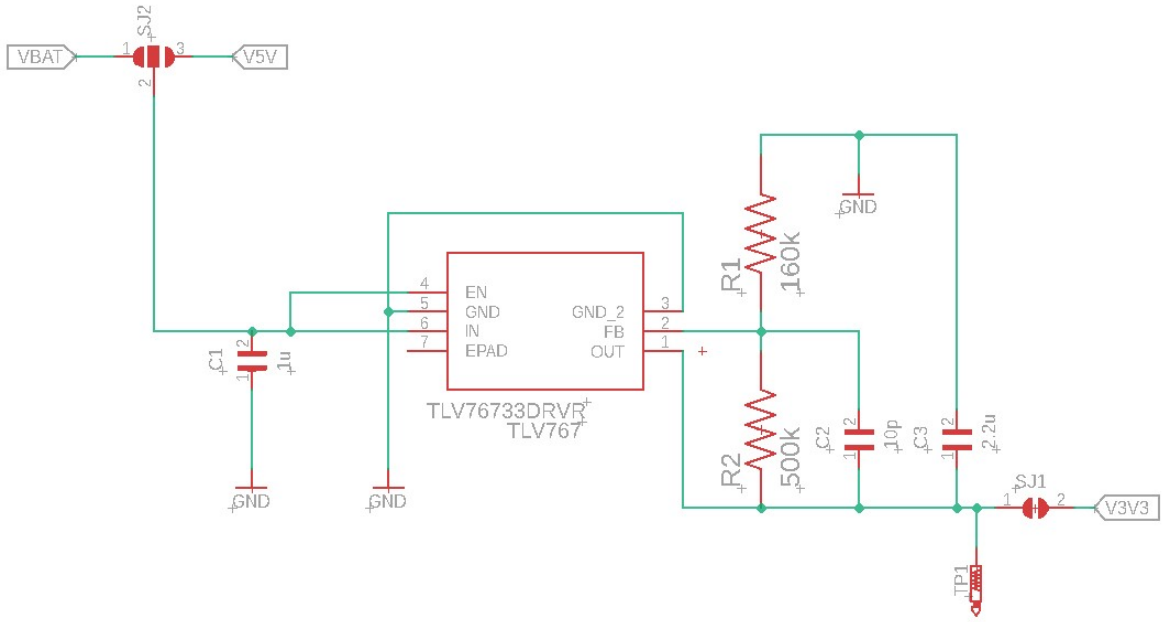

Figure 3: Power module schematic

The circuitry of the power module is shown by Figure 3. The main control parameters of the voltage regulator are the resistors R_1 and R_2, which determines the output voltage value by the equations [4]:

$$V_{out} = V_{FB}\left(1 + \frac{R_2}{R_1}\right)$$

$$R_1 + R_2 \leq \frac{V_{out}}{100 I_{FB}}$$

The desired V_{out} value is the 3.3 V to act as the main power line of the PCB, while V_{FB} and I_{FB} are known parameters with values 0.8 V and 50 nA, respectively. So, solving these equations given the known parameters yields the resistances $R_1 = 160$ kΩ and $R_2 = 500$ kΩ.

Solder joint SJ1 is a particularly important safety feature, as it initially disconnected the voltage regulator's output signal from the main power line. We needed to first perform a modular test on the power module before making the connection to the power line, since we had no guarantee as to how the voltage regulator will behave. Without the solder joint, the regulator's output signal will be directly connected to the power line, and if the voltage exceeds certain values, our components risk being destroyed by overvoltage. Once we have ensured that the voltage value is within the acceptable range of 3.3 V ± 5%, then we made the physical connection for the power line. Solder joint SJ2, though, is simply a convenience feature, to give the option of either taking power from a battery or from a constant voltage power source.

2.3.2 Sensor Module

Water flow sensor

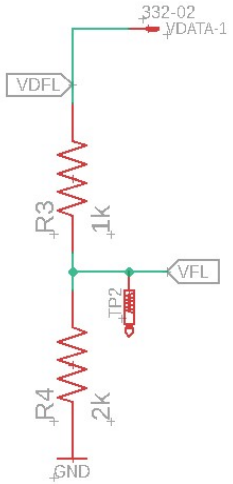

Figure 4: Flow sensor schematic

The water flow sensor that we use operates based on the Hall effect, where water flow causes the rotor to start rotating and periodically induces a voltage value [5] that matches the input voltage. The data line alternates between this value and a zero volt reading as the water keeps flowing, which means we obtain an output in the form of a square wave. By measuring the period of said square wave, we can obtain the frequency and thus the flow rate through the mathematical model:

$$F = 11Q$$

where F is frequency in Hz and Q is flow rate in LPM [6].

Our initial design was to provide the sensor with a 5 V input, but this meant the output in the form of a 5 V amplitude square wave cannot be directly read by the microcontroller since it runs on 3.3 V power (readable voltage must be <3.3 V). The circuitry required to modify the flow sensor data is shown by Figure 4, which utilizes voltage divider to tune down the amplitude to ⅔*5 = 3.3 V for the microcontroller to read. Since we have a square wave, the microcontroller can simply perform a digital read on the flow data line to obtain alternating values between logic 1 and 0 to then measure period, obtain frequency, and finally calculate flow rate using the above equation.

Current sensor

The current sensor in our system is powered on by a 3.3 V power signal and has an input current range of ±31 A with a sensitivity of 45 mV/A [7]. Input current comes in and out of IP+ and IP- ports respectively, as seen in Figure 5. Since we are only concerning ourselves with positive current measurement, we have an output voltage range of 0 V to 1.395 V, well within the value of readable voltage value using the analog read function.

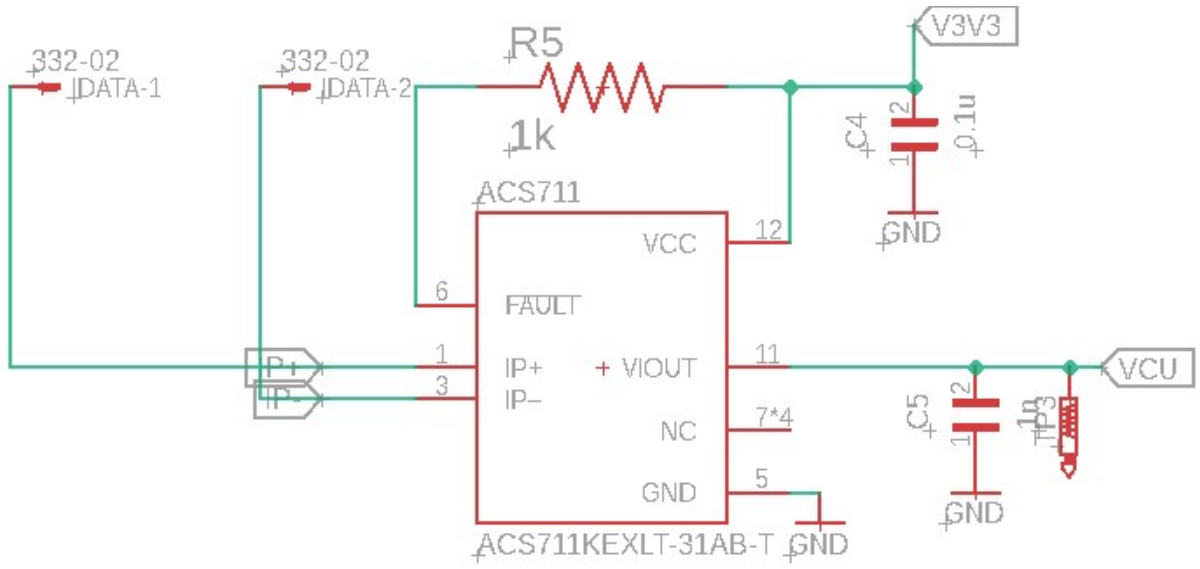

Figure 5: Current sensor schematic

Temperature sensor

The temperature sensor is powered using a 3.3 V power signal and outputs digital data. Sensor data is directly connected to a GPIO, but with a 4.7 kΩ resistor between the data and power lines. Data processing for this sensor is solely handled by a library to convert digital data to temperature value in degrees Celsius.

Vibration sensor

We are using a piezoelectric device as our vibration sensor, which means that the sensor reacts to mechanical stress and outputs a voltage oscillation with varying amplitude accordingly. There is no simple way to definitively measure a vibration, but for our implementation we are especially only concerned with an arbitrary form of measurement to be able to differentiate between normal and excessive vibration. Out of the sensor's two terminals, one is connected to ground, while the other to a GPIO to be read using the analog read function.

2.4 Software Design

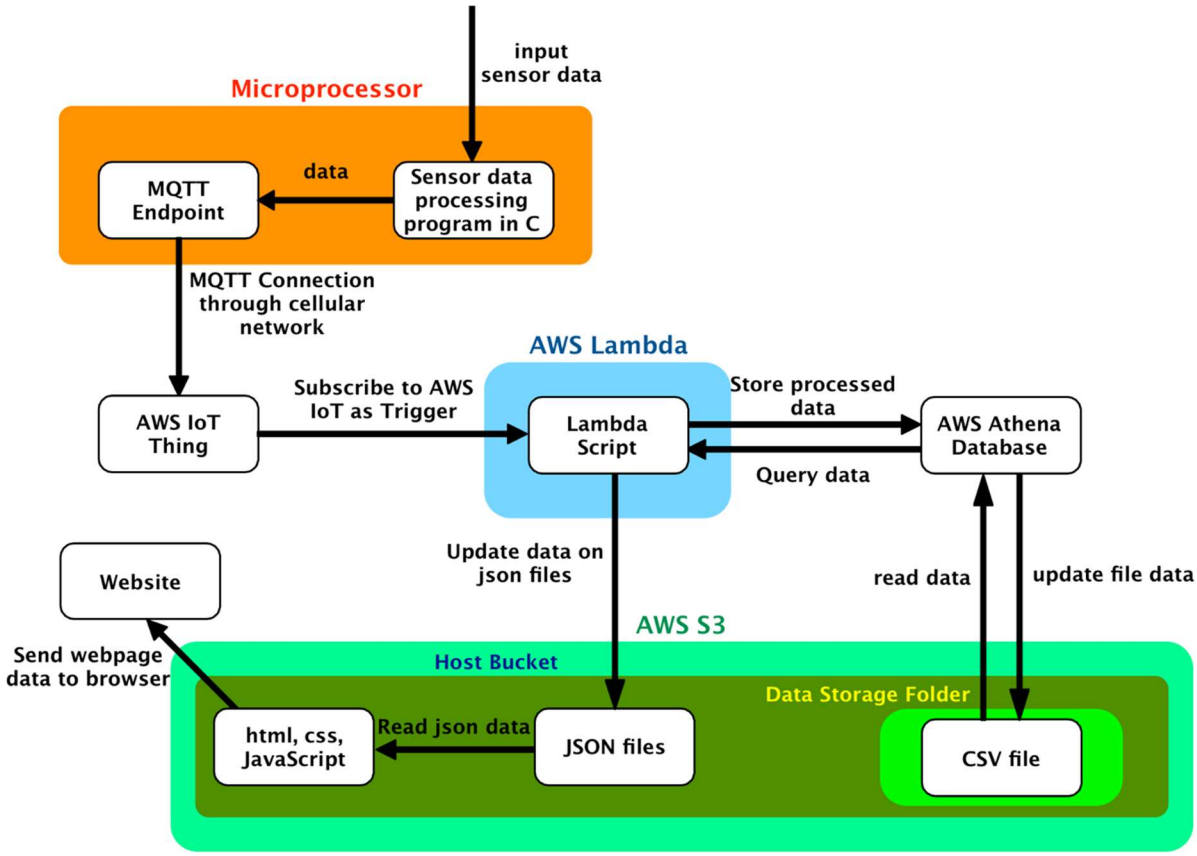

Figure 6: Block Diagram of Data Flow in Software

As represented in Figure 6 above, the functionalities of C code on ESP 32 are reading sensor data, processing the data into JSON format, and then sending data to AWS IoT through WiFi.

2.4.1 Sensor Data reading on EPS 32

ESP32 reads sensor data from GPIO pins.

Water flow sensor data is read using built-in function pulseIn(). The function pulseIn() reads HIGH or LOW pulse in microsecond. Since the flow sensor outputs constant HIGH when it is inactive, pulseIn() reads LOW in this program. The equation for flow rate calculation is as follows:

$$LPM = \frac{1}{(\,pulseIn(\,) \times 2\,) \times 1000000 \times 11}$$

Reading in seconds is needed, so the value is divided by 1000000. Single pulse is half of period, so the value is multiplied by 2 to get the whole period. Inverse is frequency, and frequency divided by 11 is LPM, which is the unit we want.

Current sensor reading is read using the default analogRead() function.

OneWire[8] and DallasTemperature[9] are used to read temperature sensor data digitally.

Vibration sensor data is calculated by averaging the 40 samples of analogRead with 25 ms intervals in between. This approach was chosen because precision is not needed, and ESP32 cannot know where the wave form starts. The 25 ms interval is chosen because it is 1.5 times larger than the period of vibration sensor signal, which is 16 ms. The sample interval is purposefully out of sync with signal period in order to avoid having large dependencies on signal phase.

2.4.2 WiFi and AWS IoT communications on EPS 32

WiFi is connected through WiFiClientSecure library. It connects to the WiFi network using WiFi name and password stored in the header file.

AWS IoT is connected through MQTTClient library. Amazon root certificate, device certificate, device private keys, and AWS IoT endpoint URL stored in header file are used to connect to AWS IoT through MQTT protocol.

Build-in ArduinoJson library is used to format sensor data into JSON, which is a suitable format for MQTT data transmission. This JSON data is published to esp32/pub IoT topic. This topic is accessible through AWS IoT services.

2.4.3 AWS Lambda

AWS Lambda script is responsible for 3 tasks: processing any incoming data from AWS IoT, storing data into the database, and updating JSON files on S3 bucket.

AWS Lambda script is triggered when AWS IoT receives data on esp32/pub endpoint. Since ESP 32 has an internal clock that starts at 0 millisecond, which is 1970 January 1st, AWS Lambda script assigns current date time to JSON data. AWS Lambda script also computes warning flags based on temperature and vibration data.

Lambda script then executes the query to insert the new data into the table. Then, data from the past 8 hours are queried. This data overwrites JSON files on S3 bucket.

2.4.4 AWS Athena Database

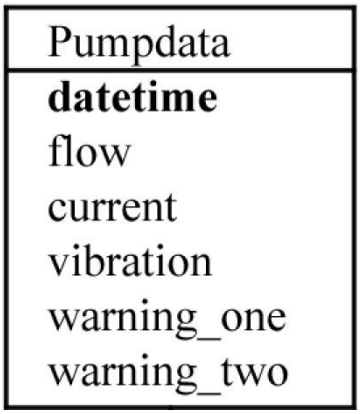

Figure 7: Database Schema

temperature	vibration	flow	electric_current	datetime	warning_one	warning_two
23.125	97.35	0.0	1879.0	2020-11-19 16:47:00.000	0	0
23.125	18.6	0.0	1881.0	2020-11-19 16:46:57.000	0	0
23.125	80.425	0.0	1883.0	2020-11-19 16:46:54.000	0	0
23.125	56.975	0.0	1881.0	2020-11-19 16:46:50.000	0	0
23.125	17.95	0.0	1881.0	2020-11-19 16:46:47.000	0	0
23.125	104.125	0.0	1877.0	2020-11-19 16:46:44.000	0	0
23.125	0.0	0.0	1878.0	2020-11-19 16:46:41.000	0	0
23.125	102.375	0.0	1881.0	2020-11-19 16:46:37.000	0	0

Figure 8: Data Stored in Database

Figure 7 shows the database schema for the table pumpdata. Datetime is the primary key, which ensures every row is unique. Figure 8 shows some experimental data stored in the database. The database stores data for four sensors, current date time, and two warning flags. Data for four sensors are in float. Datetime is in timestamp. Warning_one and warning_two are in integer, but they are used like boolean in practice.

2.4.5 AWS S3 and Website Hosting

After the Lambda script inserts and queries the newest data from Athena into S3 bucket, the frontend website reads the JSON file and outputs the content into graphs. We used ajax to unpack the contents of the JSON file, and the graphs were implemented using Google Charts.

3. Design Verification

3.1 Hardware Design Verification

3.1.1 Power Module

Table 1: Voltage measurements of power module

V_{in}(V)	V_{out}(V)	V_{FB}(V)
3	3	2.27
3.3	3.3	2.56
3.5	3.5	2.76
3.8	3.8	3.05
4	4	3.3
4.3	4	3.3
4.5	4	3.3

We performed the power module test by feeding the voltage regulator a series of increasing V_{in} values and observing the voltage values of V_{out} and V_{FB} for each case, with the results shown in Table 1. Despite expecting our V_{out} to saturate at 3.3 V, our V_{out} happens to saturate at 4 V instead, which means it cannot be used as the main power line as said value will most likely break most of our components. This error might be due to a component, specifically, a capacitor which is too small to be soldered by hand, being omitted from the circuitry.

Fortunately, the other set of data in Table 1 shows our V_{FB} saturating at 3.3 V as V_{in} exceeds 4 V, so our solution is to connect V_{FB} to the main power line instead using a wire as shown in Figure 9. Naturally, a concern arose regarding if this setup can inject enough current into the system to power on all the components. However, during the testing of the system as a whole, this power module can reliably power every component on and even provide the heaviest ~140 mA current required during data transmission.

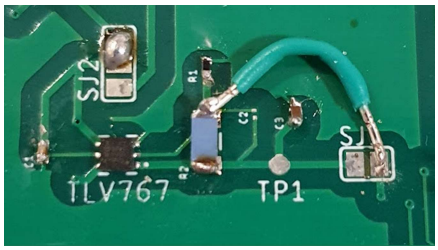

Figure 9: Physical power module on PCB

3.1.2 Sensor Module

Water flow sensor

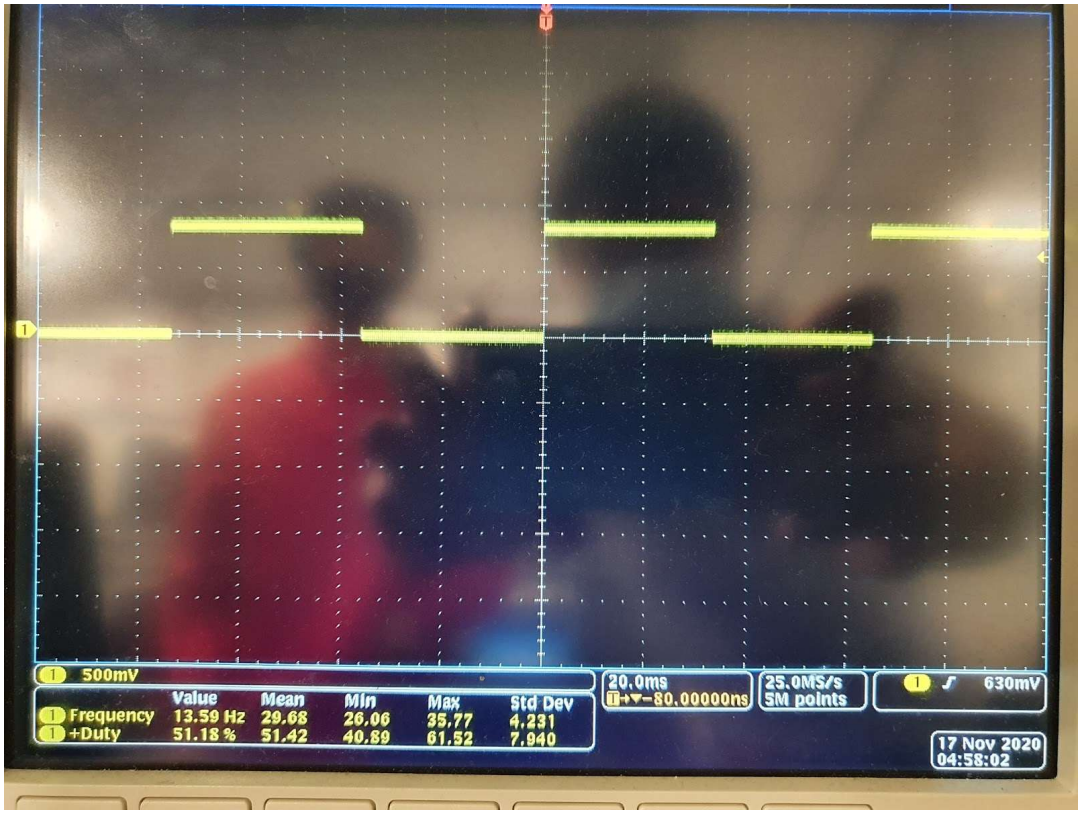

Figure 10: Oscilloscope output of flow sensor after voltage divider

During the testing stage of the water flow sensor, we found that the output of the voltage divider when water flow is present, where we expect a square wave with 3.3 V amplitude, has a 0.8 V amplitude instead as shown in Figure 10. Furthermore, we suspected that we also a little later shorted the GPIO that receives this data, such that it cannot read voltage properly anymore (outputs constant voltage value despite oscilloscope telling otherwise). We decided to go around this problem by disregarding the voltage divider circuitry and feeding the flow data directly to a GPIO with the sensor powered on by a 3.3 V power signal. The resulting data is in the original form that we expected, a 3.3 V amplitude square wave, which greatly simplifies the data processing.

Current sensor

The current sensor does not seem to work at all as it outputs a constant 1.8 V value regardless of the input. This error might be due to overheating the chip during the soldering process using a reflow oven (>200°C, while max temperature for chip is 165°C) or the usage of flux without proper cleaning, causing corrosion to pads and thus chip connections.

Temperature sensor

Testing was done by observing temperature values when it measures room temperature versus when we warm it up by holding the sensor with a hand. The sensor measures room temperature to be ~24◦C, and when we warm it up by hand the value gradually increases to ~36◦C, the typical human body temperature. Thus, the temperature sensor seems to be working well.

Vibration sensor

The vibration sensor is tested by observing its output while exposing the sensor to finger taps and phone vibrations. The larger the force applied to the sensor, the larger the amplitude of oscillation. Figure 11 shows an example of the sensor exposed to minor vibrations, with oscillation period of approximately 16 ms. This period is an important piece of information for data processing to obtain our arbitrary vibration measure.

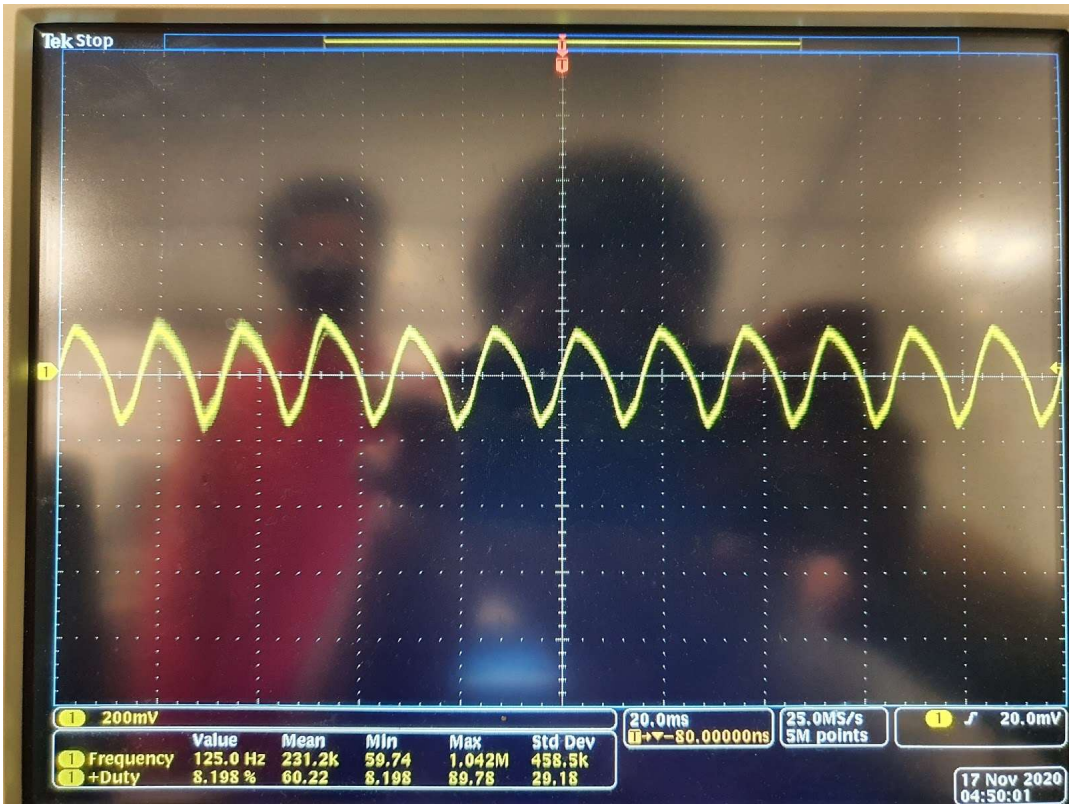

Figure 11: Oscilloscope output of vibration sensor with minor vibration

3.2 Software Design Verification

3.2.1 Sensor Data reading on EPS 32

Current	Flow	Temperature	Vibration
1841	2.39	24.12°C	595.17
1845	2.07	24.50°C	436.25
1842	0.73	25.19°C	439.48
1842	0.44	25.87°C	418.73
1843	1.20	26.50°C	421.10
1840	1.89	27.06°C	317.37
1841	1.57	27.56°C	7.70
1845	2.27	27.94°C	1545.28
1845	1.75	28.31°C	561.85
1842	1.57	28.62°C	271.62
1841	1.98	28.94°C	0.00
1847	0.68	29.12°C	64.37
1842	1.38	29.37°C	0.00

Figure 12: Sensor Readings printed on Arduino Console.

Figure 12 shows the reading of sensors printed on the Arduino Console. Current sensor is not functional, so it is outputting the same value with noise as expected. Flow sensor reading is not constant because a hand pump, which does not have a reliable constant flow rate, is used to test this sensor. This variability in result is expected and all values are within reasonable range. Temperature sensor reading here shows how temperature increases from room temperature (24°C) when the sensor is warmed with hands. Vibration sensor reading here shows phone vibration for the first 6 data points. Then the sensor is tapped with a finger which resulted in a high value of 1545.28. Our vibration sensor is more responsive to mechanical stress than to vibration, so this is the expected result.

3.2.2 WiFi and AWS IoT communications on EPS 32

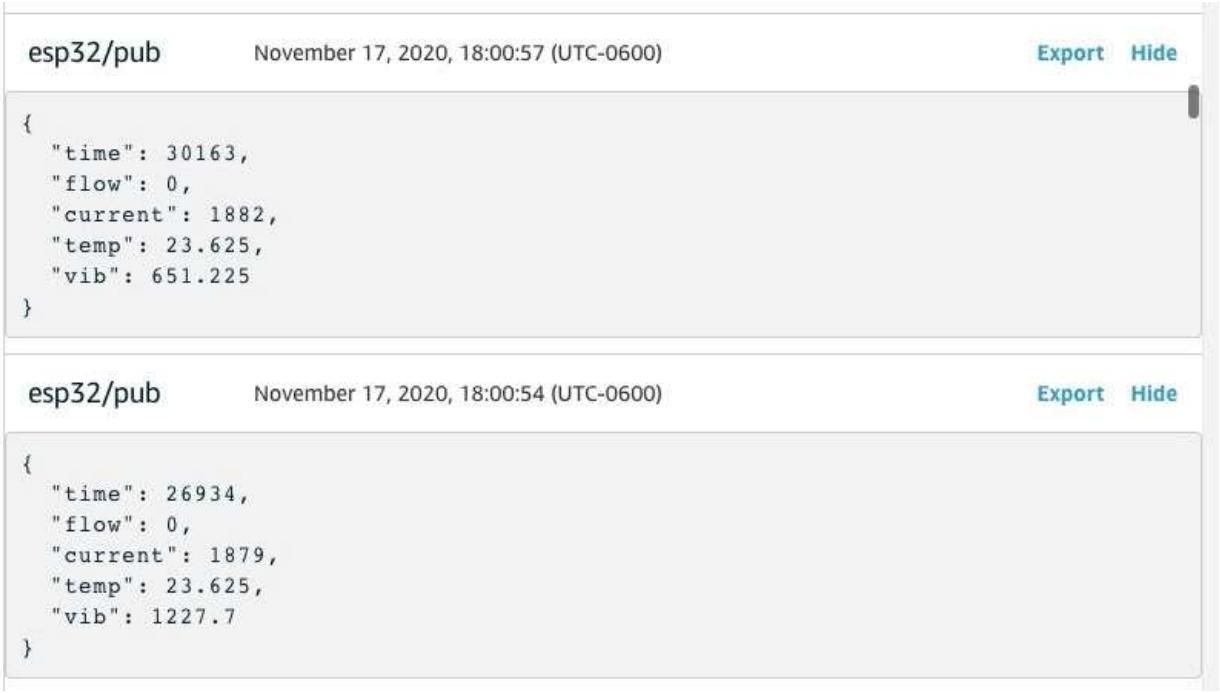

Figure 13: AWS IoT Console

Figure 13 shows the data transmitted from ESP 32 to AWS IoT console. This is the testing feature on AWS IoT console that can print out data published on certain topics. The topic esp32/pub is subscribed temporarily to show the data. Data is displayed on AWS IoT console only if ESP 32 is connected to WiFi and AWS IoT endpoint. In addition, displayed data is in correctly formatted JSON structure, which is the structure defined in C code on EPS 32.

3.2.3 AWS Lambda

Figure 14 shows the AWS Lambda Log. It shows how AWS Lambda script is executed given the mock data sent from ESP 32. The input data "event" is printed in the function. Data is successfully printed after execution, so AWS Lambda script executes on AWS IoT trigger.

Figure 14: AWS Lambda Log

3.2.4 AWS Athena

Query submitted time	Query
2020/11/19 12:09:00 UTC-5	`select * from waterpumpdata.pumpdata where datetime between TIMESTAMP '2020-11-19 09:08:56.000' and ...`
2020/11/19 12:08:57 UTC-5	`insert into waterpumpdata.pumpdata (datetime, temperature, vibration, flow, electric_current, warnin...`
2020/11/19 12:08:56 UTC-5	`select * from waterpumpdata.pumpdata where datetime between TIMESTAMP '2020-11-19 09:08:53.000' and ...`
2020/11/19 12:08:53 UTC-5	`select * from waterpumpdata.pumpdata where datetime between TIMESTAMP '2020-11-19 09:08:50.000' and ...`
2020/11/19 12:08:53 UTC-5	`insert into waterpumpdata.pumpdata (datetime, temperature, vibration, flow, electric_current, warnin...`
2020/11/19 12:08:50 UTC-5	`select * from waterpumpdata.pumpdata where datetime between TIMESTAMP '2020-11-19 09:08:47.000' and ...`

Figure 15: AWS Athena Query Log

Figure 15 shows the query log in AWS Athena database. It shows how data insert query is executed before data fetch query. Insert queries or fetch queries can be in a row, but this is due to multithreading of AWS Lambda. Since this AWS Lambda script executes in 5 seconds, script execution can overlap if the AWS IoT publishes faster than 1 data per 5 minutes. For testing purposes, the publishing interval was 1 second, so this behavior is natural and expected.

3.2.5 AWS S3 and Website Hosting

The static frontend website displays the JSON file that is stored in the S3 bucket correctly. There are four graphs that display the water pump's temperature, vibration, current, and water flow level. Figure 16 and Figure 17 were taken during the time of the demonstration. The starred points on the safety parameter graphs indicate readings that exceeded the safety threshold values.

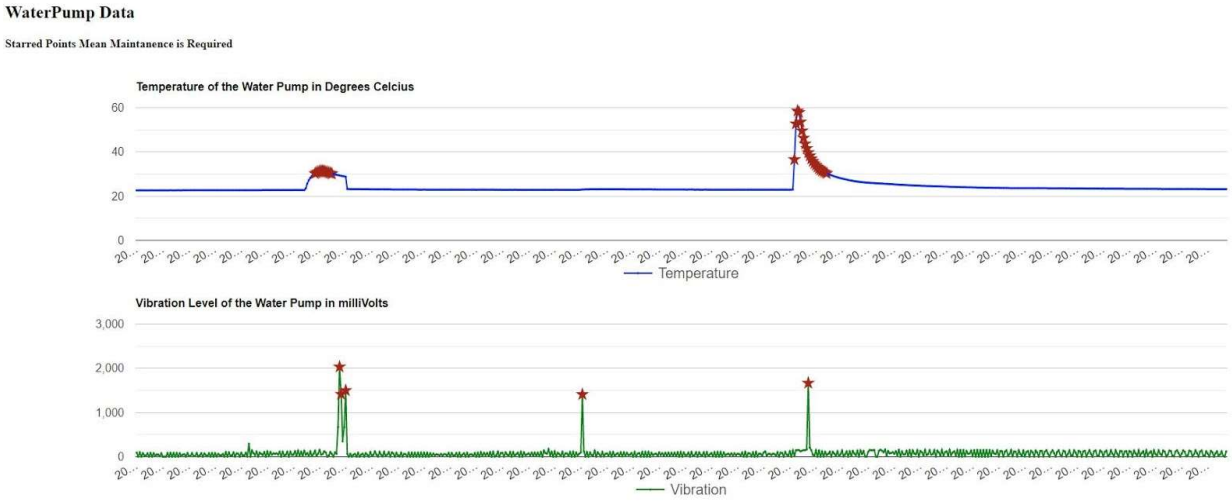

Figure 16: Temperature and Vibration Graphs from the Frontend Website

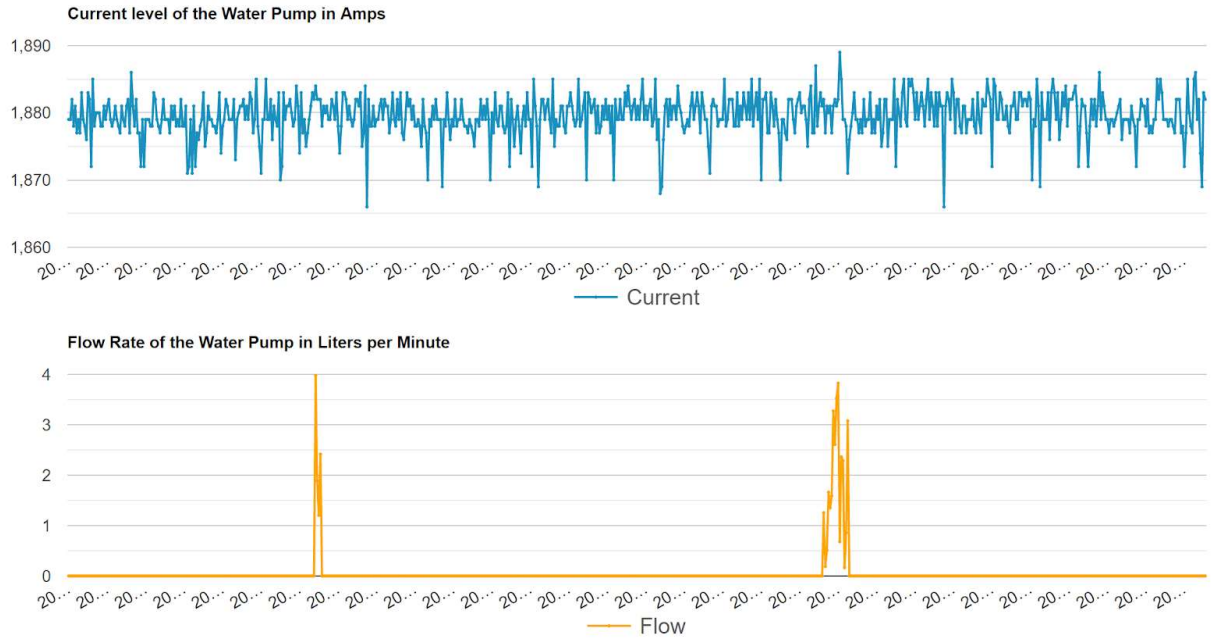

Figure 17: Current and Flow Rate Graphs from the Frontend Website

4. Costs

4.1 Parts

Table 2. Table of component cost

Description	Manufacturer	Part #	Qty	Cost (USD)
Battery	SparkFun Electronics	1568-1491-ND	1	4.95
Voltage Regulator	Texas Instruments	296-TLV76733DRVRCT-ND	1	0.83
Water Flow Sensor	Seeed Technology Co.	1597-1520-ND	1	6.02
Ammeter	Allegro MicroSystems	620-1482-1-ND	1	1.41
Thermocouple Wire	BFRobot	1738-1311-ND	1	6.97
Vibration Sensor	TE Connectivity	MSP1006-ND	1	5.37
Microcontroller	Espressif Systems	ESP32-WROOM-32E	1	2.8
Push Buttons	TE Connectivity	1825910-6	2	0.2
Header Pins	Harwin Inc.	-	52	9.03
Jumper Wires	Adafruit Industries	-	-	5.9
Resistors	Various	-	-	3.34
Capacitors	Various	-	-	1
Total				47.82

4.2 Labor

Table 3. Table of labor cost distribution

Name	Hourly rate (USD)	# of Hours	Cost (USD)
Raynaldi Yose Iskandar	35	100	3500
Masaki Sato	35	100	3500
Yun Mo Kang	35	100	3500
Total			10500

4.3 Schedule

Table 4. Summarized work schedule

Week	Task	Members **Y**un, **M**asaki, **R**ay
10/5	Sign up for Design Review, Set up AWS lambda server Test power and sensor module interface on breadboard Complete design review	M, Y R ALL
10/12	Interface with the RF transceiver Design initial frontend website Interface the sensors to the microcontroller	M Y R
10/19	Interface the transceiver to AWS server Design & finalize PCB orders	M, Y R
10/26	Connect the AWS backend to the frontend Order PCB	M, Y R
11/2	Install the sensors to the PCB Add functionalities on the frontend	R M, Y
11/9	Mock Demonstration sign up Interface everything together & complete mock demonstration	Y ALL
11/16	Complete Demonstration	ALL
11/23	Thanksgiving Break	ALL
11/30	Complete Mock Presentation Complete Presentation Write up final papers	ALL
12/7	Finish and submit final papers Lab checkout Submit lab notebooks	ALL

5. Conclusion

5.1 Accomplishments

Our monitor worked as it was expected to. The device was able to collect temperature, vibration, and water flow data then send them wirelessly to a cloud-based server. This information was then displayed as graphs with visual representations to notify users with any malfunctions of the water pumps. The time interval of data collection and display was also designed to be flexible as it can be changed with a simple change in a line of code.

5.2 Uncertainties

One main component that failed during the implementation of our project was the current sensor. We were not quite sure of the reason, but we suspect that it could have been a faulty chip, or the temperature of the soldering machine was much too high which broke the sensor. We tried to order a new sensor, but we did not have enough time. In order to fix this, we may try with new current sensors, and during the soldering process, we can try using lower temperature.

5.3 Ethical considerations

5.3.1 Ethics

This project is mainly focused to help maintain a clean water source for undeveloped regions of Indonesia. Our effort is a direct practice of the IEEE Code of Ethics #1: “To hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, to protect the privacy of others, and to disclose promptly factors that might endanger the public or the environment” [10]. Despite our efforts to assist the villagers, we realize that errors can be made, and in the case of hazardous side-effects, we swore to keep confidentiality and tend to the error immediately. This project will not only ensure clean water to the villagers but will also allow children to attend school instead of taking hour-long trips to gather water. This will enable them to pursue better lives and break the cycle of poverty. Furthermore, by the IEEE Code of Ethics #5: “to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, to be honest and realistic in stating claims or estimates based on available data, and to credit properly the contributions of others”, we realize that we are still at a learning stage, where we occasionally make mistakes and need the guidance of other, more professional personnel. Therefore, being humble and honest about our mistakes, listening to feedback, and crediting the people that helped us in the project, is a necessary step for us to improve as an engineer and, more importantly, as an individual.

5.3.2 Safety

Corrosion on the sensor modules is a serious safety concern. Since this module will be underwater, parts can corrode and dissolve into water, which can contaminate the water. The Environmental Protection Agency states that the maximum allowable for copper and lead are 1.3 milligrams/liter and 0.015 milligrams/liter, respectively [11]. This problem can be solved by building a cover that can insulate hardware from water. Any water damage to the power source is another major safety concern. Although this part will not be submerged, the weather tends to be rainy in Indonesia. Therefore, building a quality water-proof system for the monitor will be important.

The lithium-ion battery also may be a safety concern. If it is not handled properly, it may explode and cause severe damage to hardware and possibly anyone near it. Any physical damage or high temperatures above 130°F may cause damage to the battery [12]. If the battery breaks or explodes and drops debris into the water, it could become a cause of contamination. To counter this concern, we will build a casing for the batteries so that they are well-protected against external impacts and fluids.

5.4 Future work

The water pump monitoring system that we have constructed can be improved in many ways. First, the connection method that was used for our prototype was a wi-fi connection. This has many limitations when it comes to accessibility. We could improve this design by using a cellular chip instead, which was part of our original design. However, due to lack of funds and due to lack of SIM cards where payment by usage was not available, we were unable to implement our project with a cellular chip. However, we found out that the method of connection between a wi fi network and a cellular network is very similar, the only difference being that a cellular connection requires the credentials of the service provider where a wi fi connection needs the credentials of the wi fi network.

The design could also benefit from collecting additional location data. If our prototype gets mass-produced and gets placed on all the water pumps that Solar Chapter has established, it would be essential to know the location of the water pumps. Thus, we would add a small GPS that will read the longitude and the latitude of the monitor. This would then be sent to the database and, with the Google Maps API, would be displayed on a map on the website. Then the website can be improved to display individual pump's graph data when it is clicked on the map. The website will also list which pumps are in a need of maintenance along with a graphic representation on the map.

References

[1] Water.org, “Indonesia’s Water Crisis”, 2020. [Online]. Available: https://water.org/our-impact/where-we-work/indonesia. [Accessed: 25- Sep- 2020].

[2] Solar Chapter. “Water for Nibaaf.” , 2020. [Online]. Available: https://solarchapter.com/chapter/one/water-for-nibaaf [Accessed 16- Sep- 2020].

[3] G. Karina, “Project Timor Tengah Utara,” Reja Aton Energi. Sidoarjo, Indonesia. Feb. 07, 2020.

[4] Texas Instruments, “TLV767 1-A, 16-V Precision Linear Voltage Regulator”, TLV767 datasheet, Dec. 2018 [Revised Jun. 2020].

[5] Elprocus, “Water Flow Sensor Workings and Its Applications”, 2020. [Online]. Available: https://www.elprocus.com/a-memoir-on-water-flow-sensor. [Accessed: 1-Oct-2020].

[6] Seeed Technology, “Water Flow Sensor YF-B3”, YF-B3 datasheet, Jun. 2017.

[7] Allegro, “Hall-Effect Linear Current Sensor with Overcurrent Fault Output for <100V Isolation Applications”, ACS711 datasheet, Jun. 2017.

[8] “PaulStoffregen/OneWire” [Online]. Available: https://github.com/PaulStoffregen/OneWire. [Accessed: 10- Nov- 2020].

[9] “milesburton/Arduino-Temperature-Control-Library” [Online]. Available: https://github.com/milesburton/Arduino-Temperature-Control-Library. [Accessed: 12-Nov- 2020].

[10] IEEE.org, "IEEE IEEE Code of Ethics", 2020. [Online]. Available: http://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: 16- Sep- 2020].

[11] Epa.gov, “Ground Water and Drinking Water”, 2020. [Online]. Available: https://www.epa.gov/ground-water-and-drinking-water. [Accessed: 25- Sep- 2020].

[12] Osha.gov, “Preventing Fire and/or Explosion Injury from Small and Wearable Lithium Battery Powered Devices”, 2020. [Online]. Available: https://www.osha.gov/dts/shib/shib011819.html [Accessed: 28-Sep-2020].

Appendix A: Requirements and Verifications Table

Table 1. RV table for battery

Requirements	Verification
Battery can supply V_{out}>3.3V.	1. Setup a mock circuit with a 10kΩ resistor (large enough to prevent burning the resistor). 2. Connect the battery to resistor. 3. Measure voltage across resistor and confirm that V>3.3V.

Table 2. RV table for voltage regulator

Requirements	Verification
Output voltage is within the range of 3.3V±5%.	1. Power on the regulator with 5V supply voltage. 2. Measure voltage between output and ground and confirm that the value is 3.3V±5%.
Voltage regulator can output up to 1A of current.	1. Connect the output to a resistor network with ~3.3Ω total resistance. 2. Power on the regulator with 5V supply voltage. 3. Measure current and confirm that the value is ~1A.

Table 3. RV table for water flow sensor

Requirements	Verification
Functions for supply voltage 5VDC.	1. Connect sensor output to oscilloscope. 2. Connect sensor to 5V power source. 3. Pour water through the sensor. 4. Confirm that the oscilloscope shows output in the form of square waves with 5V amplitude.
Flow rate and frequency relation is described by the mathematical model F=11Q where F is frequency in Hz and Q is flow rate in LPM.	1. Connect sensor output to oscilloscope. 2. Connect sensor to 5V power source. 3. Provide a constant stream of water to the sensor for 30s, storing the water to a bucket. 4. Calculate the frequency of the output square waves from the oscilloscope data. 5. Measure the amount of water in the bucket. 6. Calculate the flow rate from the amount of water and time. 7. Check if the frequency and flow rate obtained fits the mathematical model; if not, update the mathematical model.

Table 4. RV table for ammeter

Requirements	Verification
Functions for supply voltage 3.3V±5% DC.	1. Connect sensor output to oscilloscope. 2. Connect sensor to 3.3V power source. 3. Run current through the sensor. 4. Confirm that the oscilloscope shows output in the form of voltage readings.
Current measurement is accurate up to a 5% margin.	1. Connect sensor output to oscilloscope. 2. Connect sensor to 3.3V power source. 3. Run a 15A current through the sensor. 4. Confirm that the oscilloscope shows an output voltage of 675mV/A±5%.

Table 5. RV table for temperature sensor

Requirements	Verification
Functions for supply voltage 3.3V±5% DC.	1. Connect sensor output to mock microcontroller. 2. Connect sensor to 3.3V power source. 3. Probe the air for its temperature. 4. Run a mock code to check if the sensor outputs any values.
The sensor can differentiate between pump normal operating temperature and pump overheating temperature.	1. Connect sensor output to mock microcontroller. 2. Connect sensor to 3.3V power source. 3. Probe room temperature water to simulate pump in normal operating temperature. 4. Run a mock code and note the output value. 5. Probe boiling water to simulate pump overheating temperature. 6. Run a mock code and note the output value. 7. Confirm that both readings' values are visibly distinct.

Table 6. RV table for vibration sensor

Requirements	Verification
The sensor can differentiate between pump normal operating vibration and pump excessive vibration.	1. Connect sensor output to oscilloscope. 2. Connect sensor to 3.3V power source. 3. Shake the sensor moderately to simulate pump normal operating vibration. 4. Note down the output voltage on the oscilloscope. 5. Shake the sensor harder to simulate pump excessive vibration. 6. Note down the output voltage on the oscilloscope. 7. Confirm that both readings' values are visibly distinct.

Table 7. RV table for microcontroller

Requirements	Verification
The C program on the microcontroller should be able to process data from the sensor module into a format that is suitable for transmitting data. This will be JSON.	1. Run test C program to check if signal/data from I/O pins can be recognized by the software. 2. Write test data within the C program to check if it can process data into the correct format. 3. Test the C program with sensor input to check if the data is processed correctly.
The C program can transmit data using built-in Wi-Fi capabilities to AWS server.	1. Design and run test C program in Arduino IDE to transmit mock data to AWS server. 2. Check server log of received data to confirm it matches the mock data provided in the code.

Table 8. RV table for SIM module

Requirements	Verification
The module can establish a GPRS internet connection through the 2G network.	1. Design and run test C program in Arduino IDE to establish internet connection on the ESP32. 2. Run test C program to transmit mock data to AWS server. 3. Check server log to confirm received data matches mock data.

Table 9. RV table for database

Requirements	Verification
AWS Lambda receives data given that the transmission module successfully sent data.	Send test data from hardware, and log the input received. Check if log matches with test data sent from hardware. Test this after cellular modem is tested.
Data is processed by script on AWS Lambda correctly, and processed data is stored into the database without corruption of data.	1. Create dummy data input within the script and log the processed data. Check if it is successfully processed. 2. Check database table after processed data is inserted to database. Make sure it does not affect past data, and new data is inserted without alteration.
Whenever the database is updated, AWS Lambda script should be executed to update JSON files in AWS S3 bucket.	Send a single data packet every minute and check if the csv files in AWS S3 bucket updates.

Table 10. RV table for website

Requirements	Verification
The Javascript component of the website can read and process data from csv files on AWS S3 through ajax in real time.	Upload dummy JSON file with dummy data to S3. Log the read data on console, and check if data matches with dummy data on csv.
Warnings should be generated through analysis of data read from csv files.	Create dummy JSON files. One has data satisfying warning conditions, and another does not. Test warning analysis part of code with both of csv files. Check if warnings are generated if only if the csv file has data that satisfies warning conditions.
Data and warnings can be visualised on the website.	Given the csv file, check if the website can display all data in graph and highlight the data with warnings.

Appendix B: ESP 32 C Code

```
#include "secrets.h"
#include <WiFiClientSecure.h>
#include <MQTTClient.h>
#include <ArduinoJson.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include "WiFi.h"

#define AWS_IOT_PUBLISH_TOPIC   "esp32/pub"
#define AWS_IOT_SUBSCRIBE_TOPIC "esp32/sub"

WiFiClientSecure net = WiFiClientSecure();
MQTTClient client = MQTTClient(256);

int DS18S20_Pin = 26;
int vib_pin = 39;
int amm_pin = 36;
int flow_pin = 35;
float analog_high = 0.8;

OneWire oneWire(DS18S20_Pin);

DallasTemperature sensors(&oneWire);

void connectAWS()
{
 Serial.println("Connecting to Wi-Fi");
 WiFi.mode(WIFI_STA);
 WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

 Serial.println("Connecting to Wi-Fi");

 while (WiFi.status() != WL_CONNECTED){
  delay(500);
  Serial.print(".");
 }

 net.setCACert(AWS_CERT_CA);
 net.setCertificate(AWS_CERT_CRT);
 net.setPrivateKey(AWS_CERT_PRIVATE);
```

```
  client.begin(AWS_IOT_ENDPOINT, 8883, net);

  client.onMessage(messageHandler);

  Serial.print("Connecting to AWS IOT");

  while (!client.connect(THINGNAME)) {
    Serial.print(".");
    delay(100);
  }

  if(!client.connected()){
    Serial.println("AWS IoT Timeout!");
    return;
  }

  client.subscribe(AWS_IOT_SUBSCRIBE_TOPIC);

  Serial.println("AWS IoT Connected!");
}

void publishMessage()
{
  StaticJsonDocument<200> doc;

  float vib_data = 0;

  for (int i=0;i<40; i++){
    vib_data = vib_data + analogRead(vib_pin);
    delay(25);
  }



  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);



  doc["time"] = millis();
  doc["flow"] = frequency()/11.0;
  doc["current"] = analogRead(amm_pin);
  doc["temp"] = temperatureC;
```

```
  doc["vib"] = vib_data/40.0;
  char jsonBuffer[512];
  serializeJson(doc, jsonBuffer);

  client.publish(AWS_IOT_PUBLISH_TOPIC, jsonBuffer);
}

void messageHandler(String &topic, String &payload) {
  Serial.println("incoming: " + topic + " - " + payload);

}

float frequency(){
  float period = (pulseIn(flow_pin, LOW)*2)/1000000.0;
  if (period==0){
    return 0;
  }

  return 1/period;
}


void setup() {
  Serial.begin(9600);

  connectAWS();

  pinMode(flow_pin, INPUT);
  sensors.begin();
}

void loop() {
  publishMessage();
  client.loop();
  delay(500);
}
```

Appendix C: AWS Lambda Python Code

```
from datetime import datetime, timedelta
import json
import boto3
import time

s3 = boto3.client('s3')
s31 = boto3.resource('s3')
athena = boto3.client('athena')

def lambda_handler(event, context):
    bucketname = "www.waterpumpgraph.com"
    itemname = "data/data.json"
    # obj = s31.Object(bucketname, itemname)
    # body = json.loads(obj.get()['Body'].read())

    now = datetime.now()
    yesterday = datetime.now() - timedelta(hours = 8)

    date_time = now.strftime("%Y-%m-%d %H:%M:%S.000")
    past_time = yesterday.strftime("%Y-%m-%d %H:%M:%S.000")
    print(date_time)
    print(past_time)

    insert_query = "insert into waterpumpdata.pumpdata (datetime, temperature, vibration, flow,
electric_current, warning_one, warning_two) values (TIMESTAMP '{}', {}, {}, {}, {}, {},
{})".format(date_time, event["temp"], event["vib"], event["flow"], event["current"],
int(event["temp"]>30), int(event["vib"]>1000))

    response = athena.start_query_execution(
        QueryString=insert_query,
        QueryExecutionContext={
            'Database': 'waterpumpdata'
        },
        ResultConfiguration={
            'OutputLocation': 's3://www.waterpumpgraph.com/table storage',
        }
    )

    query_execution_id = response['QueryExecutionId']
    print(query_execution_id)
```

```
    for i in range(1, 21):
        query_status = athena.get_query_execution(QueryExecutionId=query_execution_id)
        query_execution_status = query_status['QueryExecution']['Status']['State']

        if query_execution_status == 'SUCCEEDED':
            print("STATUS:" + query_execution_status)
            break

        if query_execution_status == 'FAILED':
            raise Exception("STATUS:" + query_execution_status)

        else:
            print("STATUS:" + query_execution_status)
            time.sleep(i)
    else:
        athena.stop_query_execution(QueryExecutionId=query_execution_id)
        raise Exception('TIME OVER')



    fetch_query = "select * from waterpumpdata.pumpdata where datetime between TIMESTAMP '{}' and
TIMESTAMP '{}' order by datetime".format(past_time, date_time)

    response = athena.start_query_execution(
        QueryString=fetch_query,
        QueryExecutionContext={
            'Database': 'waterpumpdata'
        },
        ResultConfiguration={
            'OutputLocation': 's3://www.waterpumpgraph.com/table storage',
        }
    )

    query_execution_id = response['QueryExecutionId']
    print(query_execution_id)

    for i in range(1, 21):
        query_status = athena.get_query_execution(QueryExecutionId=query_execution_id)
        query_execution_status = query_status['QueryExecution']['Status']['State']

        if query_execution_status == 'SUCCEEDED':
            print("STATUS:" + query_execution_status)
            break
```

```
    if query_execution_status == 'FAILED':
        raise Exception("STATUS:" + query_execution_status)

    else:
        print("STATUS:" + query_execution_status)
        time.sleep(i)
else:
    athena.stop_query_execution(QueryExecutionId=query_execution_id)
    raise Exception('TIME OVER')

result = athena.get_query_results(QueryExecutionId=query_execution_id)
result = result["ResultSet"]["Rows"]
#print(result)

result = [[y["VarCharValue"] for y in x["Data"]] for x in result]
#print(result)

cols = result[0]
result = result[1:]

result_dict = {"data": []}
for r in result:
    temp = {}
    for i, x in enumerate(r):
        temp[cols[i]] = x

    result_dict["data"].append(temp)

#print(result_dict)

uploadByteStream = bytes(json.dumps(result_dict).encode('UTF-8'))
s3.put_object(Bucket=bucketname, Key=itemname, Body=uploadByteStream)
print('Put Complete')
```