RV Power and Environment Monitoring
Final Report

Team: SDMAY22-34
Client: Thomas Daniels
Advisor: Thomas Daniels

Team Members:
Nickolas Moser - Embedded Systems Lead
Peter Rothstein - Software Lead
Kent Mark - UI/UX Implementation
Utsavee Desai - Power Systems Testing
Doug Bullock - Circuit Implementation
Michael Woo - Power Sensor Design/Testing
Jace Kunkel - Hardware Implementation
Matt McCarthy - Circuit Implementation

Project Website: sdmay22-34.sd.ece.iastate.edu
Project Email: sdmay22-34@iastate.edu

Revision: 1.0
Last Revised: 4/24/2022

Table of Contents

Executive Summary

Introduction

Of many issues RV owners face, two were identified for the scope of this project. The first issue involves the monitoring of the power consumption of the RV. While methods exist to find power usage at a given moment in time, there are no methods which can easily collect data without human interaction, and there are no solutions which store historical power usage data. The second issue involves the leveling state of the RV. As many appliances must be level on the RV to ensure proper operation, a quick, simple, and precise leveling method is desired. Most RV users complete leveling with a bubble level near the center of the RV cabin, which requires a large amount of time to travel between leveling posts, and leaves many opportunities for error.

Project Goals

There are two goals in this project: RV power usage monitoring and RV leveling information. RV power usage monitoring involves tracking the voltage, current, and power both instantaneously and historically to allow tracking of power usage trends. RV leveling information provides the angle of the RV while performing a leveling to allow a simplification of the leveling process. Both RV power usage monitoring and RV leveling information are to be provided to the user via an intuitive website interface. A high-level block diagram of the system is shown below in **Figure 1**.

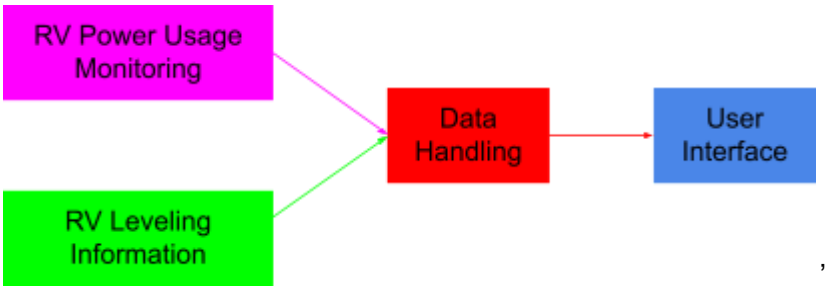

Figure 1: High Level Block Diagram of Project Goals

Applicable Courses From Iowa State University Curriculum

- CPRE288 - Embedded Systems
- EE230 - Electronic Circuits and Systems
- EE303 - Power Electronics and Energy Systems
- SE309 - Software Development Practices
- SE310 - Construction of User Interfaces

New Skills and Knowledge Acquired

- Current sensing designs
- AC voltage monitoring designs
- TCP/IP networking and data transmission
- Sleep mode integration for microcontrollers
- Setting up a reverse proxy for our web server
- Reading and handling incoming data from multiple devices through a serial connection
- Remotely sending and receiving data between pi zero and server through post and get requests

Team

Nickolas Moser is a Senior majoring in Electrical Engineering with a focus in VLSI. Roles for this project include embedded systems lead, communication facilitator, and documentation lead.

Peter Rothstein is a Senior majoring in Software Engineering, with fields of interest in software testing, automation, embedded systems, and full stack development. Roles for this project include software lead, developer, and software integration engineer.

Kent Mark is a Senior majoring in Software Engineering with experience in both backend and frontend development. Roles for this project included UX design, documentation facilitator, and communication facilitator.

Doug Bullock is a Senior majoring in Electrical Engineering with fields of interests in circuits, PCB design, and embedded systems. Roles for this project include circuit implementation, circuit testing, and component configuration.

Utsavee Desai is a senior majoring in Electrical Engineering with focus on Power Systems and Control systems. Roles for this project included circuit design and circuit testing.

Michael Woo is a senior majoring in Electrical Engineering with an emphasis on VLSI and Power Systems. Roles for this project included circuit design for power sensors as well as testing and calibration.

Matthew McCarthy is a senior majoring Electrical Engineering with a focus in Power Systems. Roles for this project included circuit testing and circuit implementation.

Jace Kunkel is a senior majoring in Electrical Engineering with an emphasis in power and energy distribution. Roles for this project included circuit testing and implementation.

Project Requirements

Problem Statement

There are two problems RV users face which this project aims to solve. The first issue relates to the power usage of the RV. While the RV can consume a large amount of power by running various appliances, there are few low-cost and/or user-friendly methods to measure the instantaneous power which is consumed by the RV. Additionally, there are no known ways to automate the tracking of RV power consumption data. As the RV contains both a battery-powered DC system and generator/shore power AC system, there are a few different power sources on the RV which can be metered. The first goal of this project is to develop a power metering system which can measure the amount of generator/shore AC power and battery-powered DC power which is consumed by the RV. This would allow the user to view the

amount of power used either instantaneously or over a period of time for the various power systems within the RV.

The second issue relates to the leveling of the RV. As many appliances cannot operate when the RV is not properly leveled, the ability to orient the RV correctly is desired. This is typically performed using a bubble level placed in the center of the RV cabin, which can leave room for error. Leveling may also take a large amount of time, as the user must walk from a leveling post to the bubble level to check whether adjustments are needed. The second goal of this project is to provide the user with a wireless interface through which leveling data can be obtained. This will eliminate the need to return to the RV cabin to check a bubble level, increasing the speed in which the RV can be leveled. Additionally, various sensors can return angle/vector data which can offer a significant improvement in precision compared to a bubble level, reducing the probability of errors in the leveling process.

Functional Requirements

Quantitative requirements are as listed below:

- The system shall provide measurements of DC voltage accurate within 5mV.
- The system shall provide measurements of DC current accurate within 10mA.
- The system shall provide measurements of DC power accurate within 50mW.
- The system shall provide measurements of AC voltage accurate within $10mV_{RMS}$.
- The system shall provide measurements of AC current accurate within $10mA_{RMS}$.
- The system shall provide measurements of AC power accurate within $100mW_{RMS}$.
- The system shall provide leveling angles accurate within +/- 0.5 degrees.
- If queried, the system shall collect AC and DC measurements instantaneously.
- If not queried, the system shall collect AC and DC measurements once per minute.
- The system shall store one month of relevant AC and DC measurements.

Qualitative requirements are as listed below:

- The website’s user interface shall be intuitive and easy to navigate.
- The website’s user interface shall be accessible from anywhere with internet connection.
- The system shall be designed to reasonably minimize power consumption.
- The system shall be reasonably installable within various RV models.
- The system shall be able to collect AC and DC measurements without user input.
- The system shall update AC and DC measurements without user input.
- The system shall update leveling angles without user input.
- The server shall be up and running 24/7.

Design Constraints

The design of the system has constraints as listed below:

- The total cost of components should not exceed $400.
- Power sensor systems should be capable of withstanding internal RV temperatures not exceeding 120°F.
- All systems should be capable of withstanding mechanical stress and vibrations associated with RV travel and use.
- All system components should be as unobtrusive as possible to the end user.

- The power consumption of the end product shall be a minimal percentage of total RV power consumption.
- Time associated with design, testing, and implementation is limited to the time frame necessitated by the course.

Relevant Standards

Relevant standards for this project include:

- IEEE 802.11
    - IEEE 802.11 sets a variety of standards and protocols for local area networks. As the project will be utilizing wireless components which communicate with a server on the same local area network, this standard will dictate throughput of measurements from wireless components to server, channels/frequencies available to the devices, and dataframes for transmitted information.
- ANSI C84.1
    - ANSI C84.1 is a high voltage classification standard which is to be observed with the project. This standard establishes voltage ratings and operating tolerances for 60Hz AC systems above 100V. With this standard, voltage classifications for the systems are given for the project along with relevant information for the associated voltage classifications. Any modification to the RV's electrical system should not violate this standard or change the classification of the RV.
- OSHA 1910.303
    - OSHA 1910.303 details safety measures and work standards for the project. Within the context of the project, it outlines safety standards involved with the design and installation of electrical systems within the RV. While OSHA is typically intended to implement safety standards between an employee and employer, we found the safety standards described with this standard to be relevant to both the design and installation of the project's electrical systems.
- AEC-Q100, AEC-Q101, AEC-Q200
    - AEC-Q100, AEC-Q101, and AEC-Q200 describe stress test qualifications for integrated circuits, discrete semiconductors, and passive components for use in automotive applications, respectively. As an RV is a motor vehicle, electronics which are used in the project shall either meet this qualification or be protected in such a way that the risks associated with automotive-based stress are mitigated.

Technical Complexity

The technical complexity of this project can be divided into four categories.

1. **AC and DC Power Monitoring**: While solutions exist for both AC and DC current, voltage, and power monitoring, none exist within the context of the project. This will involve the creation of sensors which take advantage of physical phenomena to convert various quantities to voltage measurements that can be interpreted by the Raspberry Pi Pico. Safety considerations must be made to avoid damage to power systems which could cause RV malfunctions or expose users to dangerous voltages. Standards involving stress on components must also be considered for robust system design.

2. **Microcontroller and Server Interaction**: As the Raspberry Pi Pico is a newer microcontroller, there is a lack of documentation for systems which take direct advantage of a wi-fi coprocessor to transmit data from microcontroller to server. Existing methods can be adapted to fit the scope of the project, but require significant effort to meet project requirements. Additionally, as sensor power consumption is to be minimized, methods to cut power usage at the microcontroller level without interfering with server communications are necessary. Sleep modes exist, but can either significantly reduce responsiveness or add additional complexity to the code base. Robust testing is necessary to prove results of the power savings methods as well.
3. **Data Storage and Handling**: As our project will be collecting power usage data once per minute and should store a month's worth of data, we will need a system to manage the 40,000+ data points that can be collected over the course of a month. Data will need to be stored and processed in such a way that it can be accessed by the user with minimized latency without data loss. While a database may be an easy solution, our data storage is limited to the storage available on the Raspberry Pi Zero W.
4. **User Interface**: Without an intuitive user interface, the user will have no means to view power consumption or leveling state. Designing user interfaces always comes with challenges, as different users have different opinions on how they prefer to interact with various sources of information. While there are general "best practices", our project will necessitate feedback in the development process to deliver a robust user interface.

Improvements from Existing Solutions

While solutions exist to monitor the voltage and current in a given AC or DC system, they do not meet the requirements set forth in the project. While these sensors can return instantaneous data, they require the user to physically take the measurements, which eliminates the convenience of wireless monitoring and adds safety issues when dealing with high voltage AC systems. Additionally, these systems cannot be used to store historical data, only offering measurements taken at the moment in time in which they are collected, passing the burden to the RV user to make note of power consumption over time if they are to keep track of historical trends. Lastly, there is a large cost barrier for these tools, with costs extending into the sub-$1000 range for brand-name products. A typical RV owner will likely not justify the cost of these measurement tools if they are not capable of performing measurements without the user present or an inability to store historical data.

In researching, the group found no method or product which provided wireless leveling data. Digital levels were found, but existed within the $250 range. The components utilized for our leveling system have a combined price that does not exceed $50, which offers a significant cost reduction. Additionally, digital levels often require calibration before use. While our leveling system also requires calibration before use, it can be guaranteed that the leveling system is calibrated in relation to the RV. If the digital level is not specifically used for the RV, it must be re-calibrated before use in the RV, adding additional time to the leveling process.

As discussed above, our solutions will offer an improvement in both cost and convenience from existing solutions. RVs are inherently expensive, so a low cost solution would likely be appealing to a majority of RV owners. Additionally, RVs are difficult to maintain, and data returned by our system can be used to troubleshoot simple issues which would normally

not be accessible without a visit to an RV service center. Lastly, as one purpose of an RV is to provide the user with an opportunity to enjoy traveling, camping, or other outdoor activities, an increase in the convenience associated with utilizing an RV will contribute to the user's enjoyment.

Design Overview

High-Level Overview

The system implemented for our solution consists of four main subsystems: **Power Sensors, Embedded Systems, Data Handling,** and **User Interface**. The power sensors serve to convert physical AC and DC voltage and current measurements into voltage measurements which can be interpreted by an analog-to-digital converter on a microcontroller. The embedded systems serve to interface the power sensors and IMU to the data handling system. The data handling system serves to query AC and DC voltage and current measurements, compute AC and DC power, store relevant measurement data, and provide measurement data to the user interface. Lastly, the user interface provides information stored within the data handling system to the user. A high-level block diagram of the design is shown below in **Figure 2**.

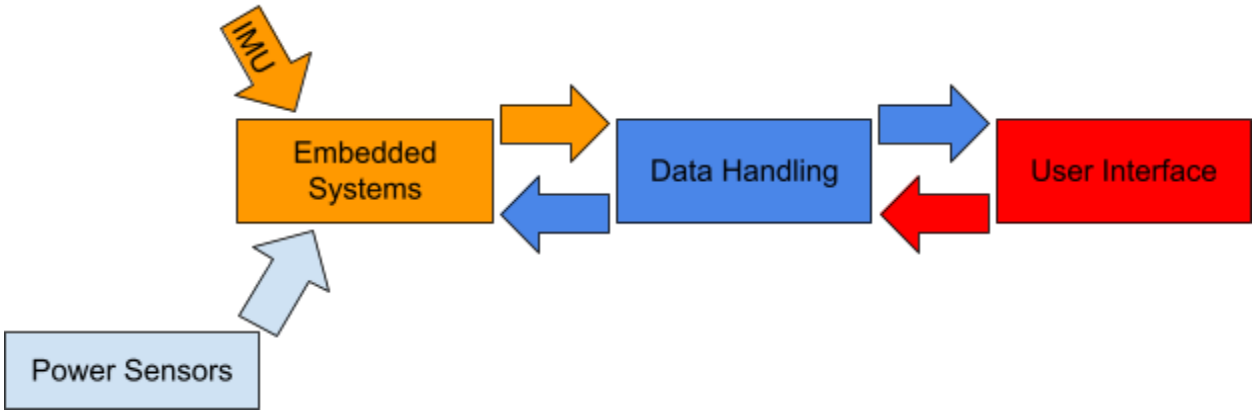

Figure 2: High-Level Block Diagram of System Design

Power Sensors

The power sensors for the RV focused on monitoring the DC current and voltage from the 12V batteries, as well as the AC current and voltage from the generator onboard the RV. The AC current and voltage monitoring circuits were stored in an outlet box identical to the one that can be found in the RV. We then connected our outlet box in series with the one in the RV, this way we could monitor the AC power without tampering with the generator, and our sensor box could be installed by just plugging the power for the RV into our outlet box, and our outlet box into shore power. For our DC sensors, we were unable to complete a design for how we would attach the sensors to the battery. The specific designs for each sensor are listed below:

1. **DC Voltage**: The design for the DC voltage circuit involved directly connecting to the terminals of the battery and then using a voltage divider to reduce the voltage to suitable range for our ADC. The voltage was then buffered using an op-amp in unity gain configuration. The output voltage from the op-amp was then fed directly to the ADC of the Raspberry Pi for data collection.

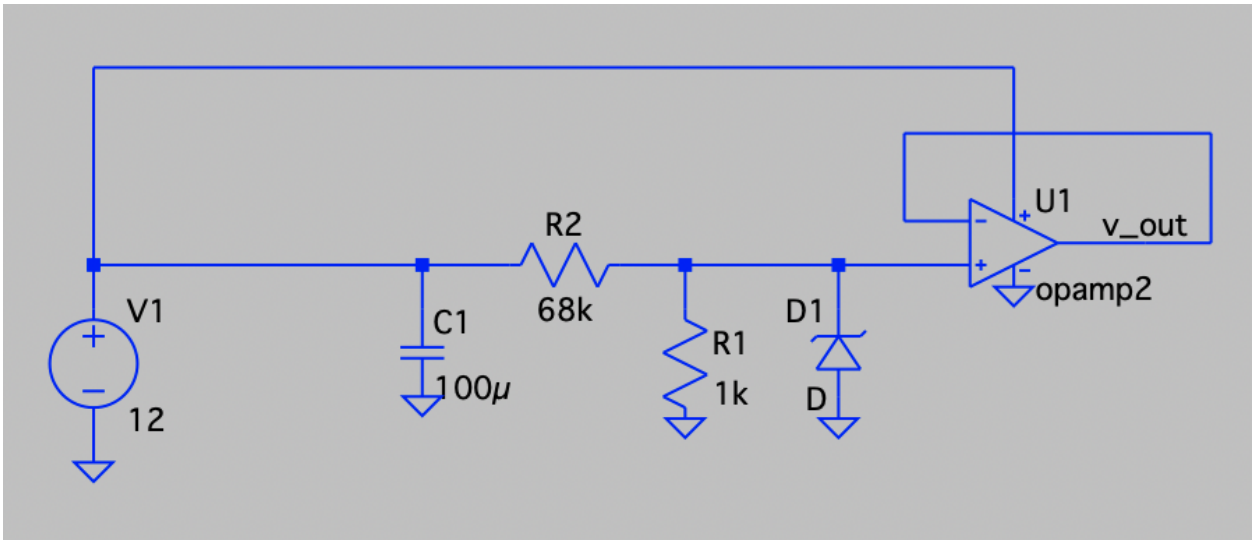

Figure 3: DC Voltage Sensor Architecture

2. **DC Current**: The design for the DC current circuit involves using a shunt resistor and a micropower instrumentation amplifier to monitor the current coming from the battery. Output current from the shunt is sent to the ADC for data collection.

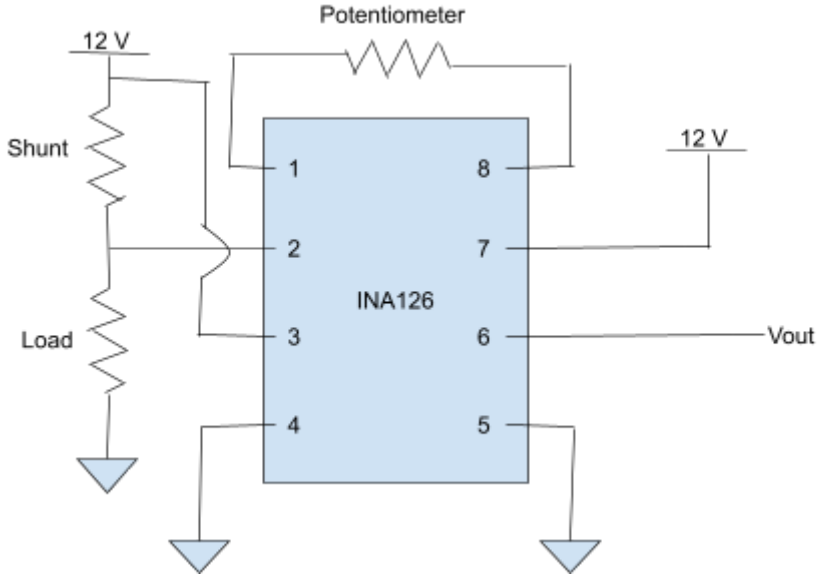

Figure 4: DC Current Sensor Architecture

3. **AC Voltage**: The design for the AC voltage monitoring was nearly identical to the DC voltage monitoring circuit. The lone addition to the circuit architecture was a high voltage power rectifier that was then fed through a similar voltage divider. The resistors used in the voltage divider were made sufficiently large to compensate for the higher voltage of the generator relative to the battery. The voltage across the divider was then buffered using an op-amp in unity gain configuration, and the output voltage from the op-amp was fed directly to the ADC on the Raspberry Pi for data collection. The general architecture for the circuit is shown below:

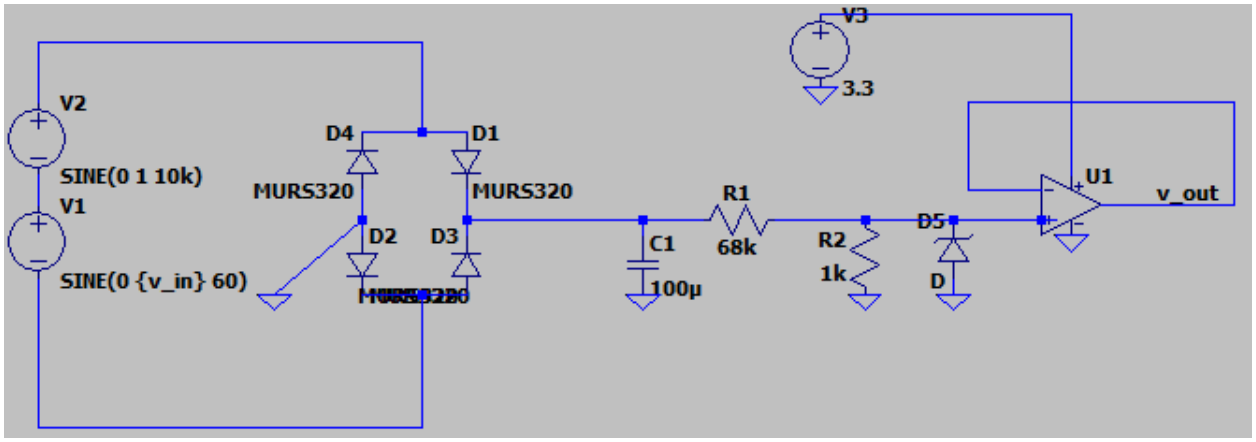

Figure 5: AC Voltage Sensor Architecture

4. **AC Current**: The design for the AC current monitoring circuit was based heavily on using a current transformer to isolate and step down the current coming directly from the generator. By using a current transformer, we never directly needed to splice into any of the wiring of the generator, and could set up a simple circuit using a load resistor and rectifier to generate a signal that could be fed to the ADC on the Raspberry Pi for data collection. The circuit architecture is shown below. The current source is symbolic for the secondary current generated by the current transformer.

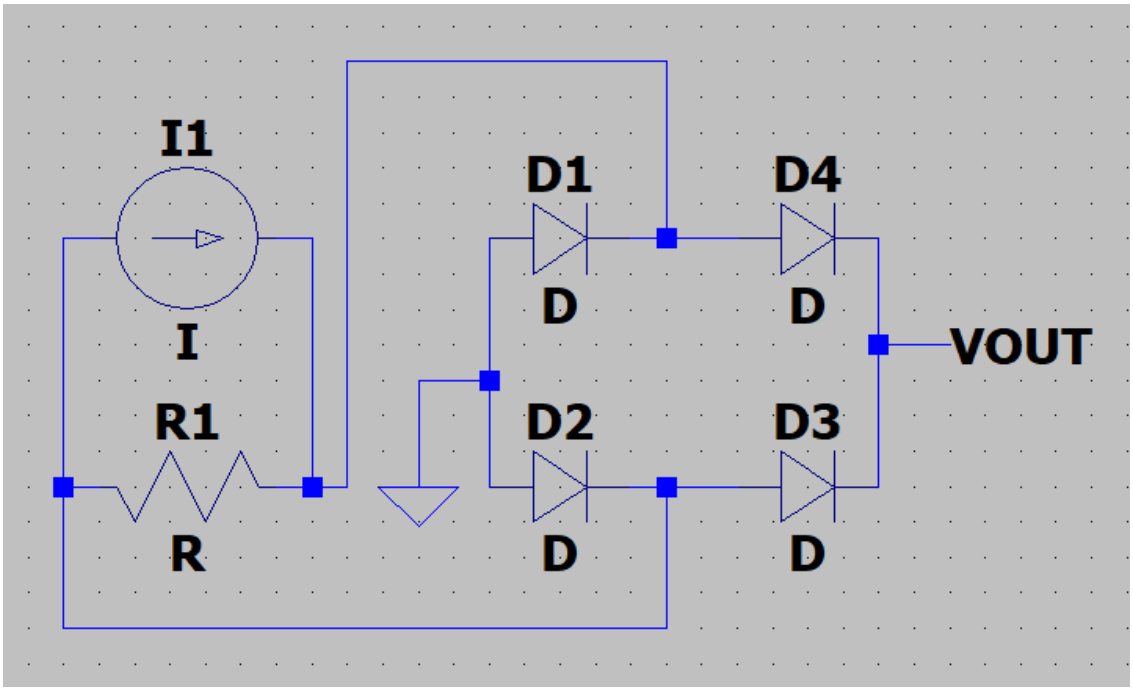

Figure 6: AC Current Sensor Architecture

Embedded Systems

The embedded systems portion of the project serves as the interface between sensors utilized within the project and the data handling system. Relevant sensors include the power sensors discussed in the previous section and the inertial measurement unit (IMU) used to perform leveling of the RV. The power sensors are interfaced via a Raspberry Pi Pico with an ESP8266 wi-fi coprocessor which wirelessly transmits data to the Raspberry Pi Zero W. There are two Raspberry Pi Pico's present in the design: one for DC power sensor measurements and one for AC power sensor measurements. In contrast, the IMU is interfaced directly via the Raspberry Pi Zero W. All devices involved will be mounted within 3D printed enclosures intended to minimize mechanical and thermal stress associated with RV usage. Code was written in python for all portions of this subsection. A block diagram of this system is shown below in **Figure 7.**

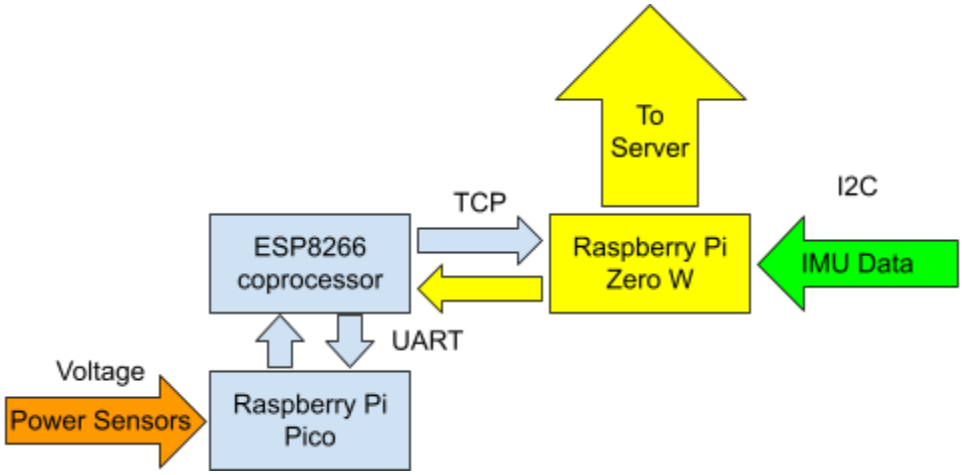

Figure 7: Block Diagram of Embedded Systems

The Raspberry Pi Pico was utilized mainly for its 12-bit ADC. While Arduino products are a common choice for embedded systems projects for their ease of use, they typically only have 10-bit ADCs, which have 1/4th of the resolution of the Raspberry Pi Pico's ADC. Additionally, as Arduino products typically operate at 5V, their ADC resolution is degraded compared to the Raspberry Pi Pico, which operates at 3.3V. For example, a 10-bit ADC with a reference voltage of 5V offers a resolution of 4.88mV/step, whereas a 12-bit ADC with a reference voltage of 3.3V offers a resolution of 0.8057mV/step, providing roughly 6 times more accuracy per step. While using the Raspberry Pi Pico added complexity due to reduced user documentation, it increased our ability to provide the user with accurate data on power consumption.

The ESP8266 wi-fi coprocessor was utilized for this project as a previous design did not provide a robust solution to data transmission between Raspberry Pi Pico and Raspberry Pi Zero W. This previous solution is discussed in **Previous Designs / Changes from 491**. While this component adds complexity to the design, it serves as a capable intermediary node between the Raspberry Pi Pico and Raspberry Pi Zero W. In operation, the ESP8266 receives queries from and sends responses to the Raspberry Pi Zero W via a TCP socket connection and the AT command set. The ESP8266 hands off queries to and transmits data from the Raspberry Pi Pico via UART. Data transmission and reception is limited to 2 kilobytes on the ESP8266, but this proved sufficient for our application.

While newer wi-fi coprocessors exist, the ESP8266 was used due to a lack of availability of newer wi-fi coprocessors. The AT command set is a set of proprietary commands which are used to interface as a modem in wireless applications. This command set was used as it is packaged with the default ESP8266 firmware, and updates which allow other programming languages could not be made due to a lack of availability of hardware interface ICs.

Upon startup, the Raspberry Pi Pico runs various setup commands to initialize the ADC and set up connections with the Raspberry Pi Zero W. The wireless setup process consists of sending AT commands to the ESP8266 via UART to query firmware version and MAC address, connect to a wi-fi access point connection, and create a TCP socket connection with the Raspberry Pi Zero W. Once a TCP socket connection is established, the Raspberry Pi Pico waits for the Raspberry Pi Zero W to send a query for measurement data.

Once a measurement query is received, the Raspberry Pi Pico begins taking ADC readings across two ADC channels and placing them in a 2 kilobyte buffer. Each power sensor is assigned to an ADC channel in a given AC or DC power system. Tags are appended to the measurement depending on the power sensor and associated ADC channel in which the measurement originates. Extra characters are added to allow parsing between tags and readings on the Raspberry Pi Zero W. **Figure 8** shows an example DC voltage measurement with tag and parsing characters.

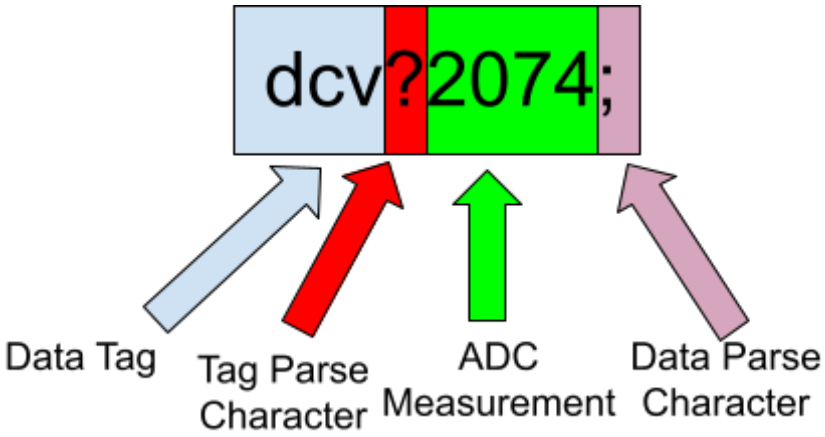

Figure 8: Example DC Voltage Measurement with Tag and Parsing Characters

Once the 2 kilobyte buffer is filled to a set threshold to prevent overflow, an end tag is appended. Data in the buffer is then transmitted to the Raspberry Pi Zero W via the established TCP socket connection on the ESP8266. The Raspberry Pi Pico will wait until it is queried again for measurement data and repeat the above process. **Figure 9** shows a state diagram of the Raspberry Pi Pico's operation.

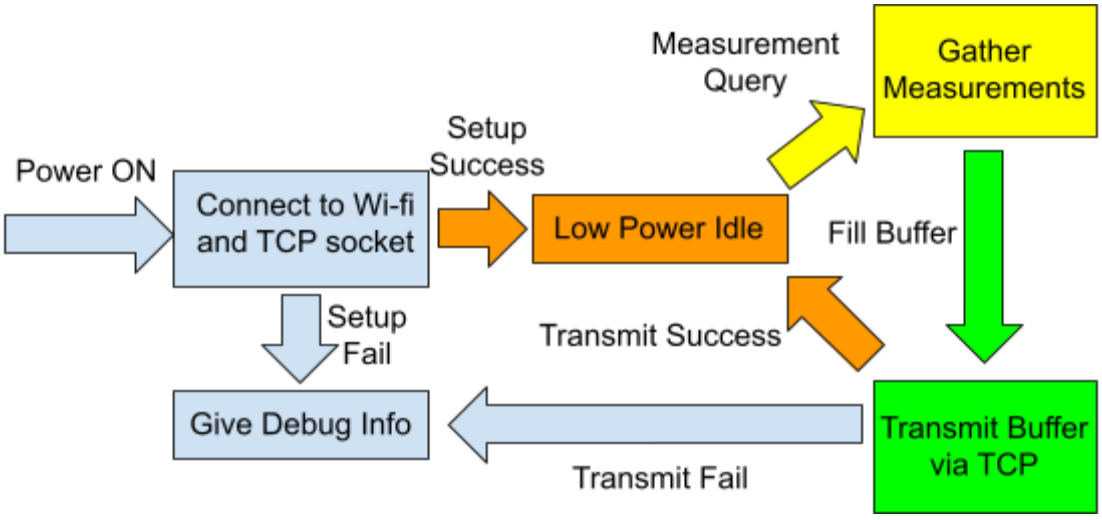

Figure 9: State Diagram of Raspberry Pi Pico Operation

While waiting for a measurement query, the Raspberry Pi Pico will enter a low power idle state. This is accomplished by maximizing clock speed while gathering measurements and transmitting data, and minimizing clock speed while idling. During setup, or when a measurement query is received, the clock speed increases to 250MHz to ensure sufficient responsiveness and data acquisition. When setup is completed, or data transmission is completed, the clock speed is reduced to 16MHz. It is shown in **Design Testing: Embedded Systems** that reducing the clock speed during idling can provide an ~80% reduction in power consumption during operation.

The IMU utilized for the project is the ICM-20948 9-DoF IMU breakout board manufactured by Adafruit Industries. This device offers a simpler hardware interface for the ICM-20948 IMU manufactured by InvenSense, with an onboard power supply, breadboard-friendly I2C pinout, and level shifting. In addition to the user-friendly hardware interface, Adafruit Industries also provides a python library for simple I2C interaction with the breakout board. This aided in the simplification of development for RV leveling angle measurement.

When implemented, the IMU will be fixed within an enclosure to ensure that the position of the breakout board remains as constant as possible. This was accomplished via a 3D printed enclosure, shown below in **Figure 10**. The enclosure allows space for both the Raspberry Pi Zero W and IMU, and leaves an opening for ports on the Raspberry Pi Zero W. Additionally, a lid can be installed if the user does not desire to view the components once they are mounted.

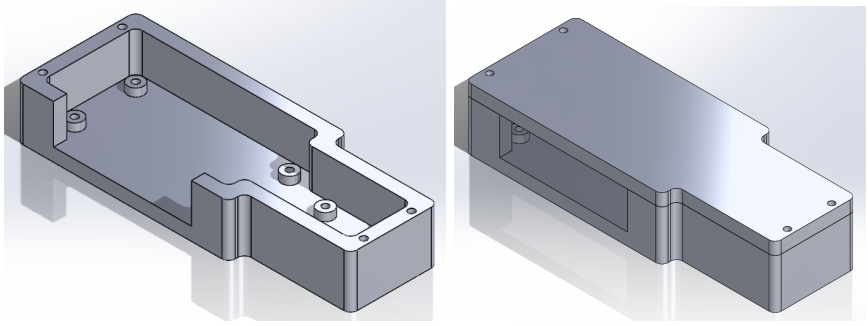

Figure 10: Simple Enclosure for IMU and Raspberry Pi Zero

Although the IMU provides accelerometer, gyroscope, and magnetometer data, only the accelerometer data is used for this portion of the project. When queried, the accelerometer provides acceleration in the X, Y, and Z directions. Using the atan2() function provided within python, the acceleration vectors in the X, Y, and Z directions can be converted to angles which describe the leveling of the RV from front to back and side to side. **Figure 11** shows an example of this vector to angle conversion.

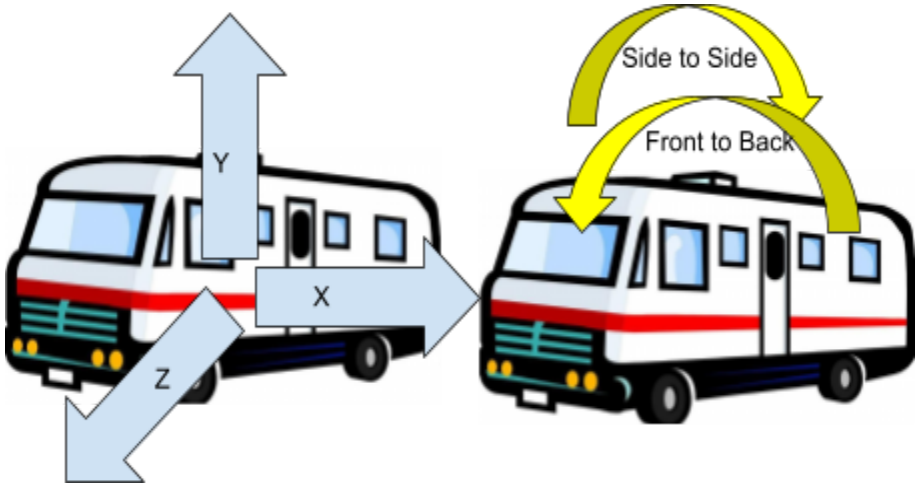

Figure 11: Vector to Angle Conversion for RV Leveling

There are two interfaces which interact with the IMU: a calibration interface and a leveling interface. The calibration interface is necessary for proper function. As the IMU returns angles relative to the X, Y and Z vectors of the IMU breakout board, it is not accurate relative to the RV if the wall on which it is installed is not exactly parallel to any of the IMU breakout board's X, Y and Z vectors. Once a user completes leveling manually, they are able to press a "calibrate" button within the user interface. The calibration interface takes a large amount of measurements to establish its installed orientation relative to the IMU breakout board's orientation. These orientations are saved on the Raspberry Pi Zero W and utilized within the leveling interface. While little drift was observed in testing, occasional calibration may be necessary depending on the method used to install the IMU enclosure within the RV. If the IMU enclosure is not well secured, its angle relative to the wall may change during travel or use, making the previously calibrated angles incorrect.

The leveling interface returns the angles needed to level the RV to the previously calibrated leveling angles. This leveling angle is then provided to the user via the user interface for use in leveling the RV. While the IMU is capable of calculating leveling angles almost instantaneously, a short delay is incorporated in the code during this process. As RV leveling is a relatively slow process, normally taking 10-15 minutes, returning an angle every second or every couple of seconds works to save power during usage while still providing the user with sufficient information. Taking 50+ measurements per second does not add any value during the leveling process, and only serves to consume unnecessary power. Once leveling is completed, the leveling interface can be closed, and the system will not query leveling data until the interface is reopened.

Data Handling

To handle incoming data, we decided to design a system that allows us to effectively read both AC and DC readings in separate scripts. Since a zero has two UART channels, we decided that instead of using one serial connection for both incoming pico readings, we would open two separate serial ports in two separate python scripts, and divide them by AC and DC readings.The serial interface was used due to a lack of time to integrate the TCP socket protocol discussed in the previous section. Both systems essentially function the same; the only difference is the communication protocol. The general design for each script is as follows:

- Open a serial port, either ACM0 driver for AC, or ACM1 driver for DC
- Sort incoming data by a given tag associated with the reading, for example “acv”, being an AC voltage reading.
- Parse and split the data by a planted ‘?’ between tag and value
- Setup a POST request to query a set function for whatever data is gathered
- From there, read in the data and display on the user interface

This is the simple overview, but in the background we are also checking for table cleaning conditionals, checking for an instant history button click to be set to true, and power calculations occurring after voltage and current readings. This is where there is a lot of post and get requests occurring at once, and where some timeouts needed to be implemented in order for our data to be sent and received without error.

Figure 12: Data communication/integration setup

Above in **Figure 12** shows what we designed as our data transfer system using two pi picos and a pi zero. To the right, you will see the zero and an IMU within a 3D printed enclosure. This is what we are using when collecting and calculating our leveling data. To the left you see the two picos which are connected to a USB hub, which is connected to the zero, where data is sent simultaneously through two separate UART channels.

User Interface

The website as a whole was designed to be a low-level interface that is accessible from anywhere. In order for it to be accessible from anywhere, and not just on the Iowa State campus, we had to set up a reverse proxy to an external-facing hostname. This means that when hosting our web server, this proxy will sit in front of it, forwarding client requests to our web server, acting as a filter and a gateway to a public IP, accessible from anywhere. Outside of networking, our design consisted of the following:

- GitLab: source control
- Git: version control
- React: frontend
- Node.js: backend
- MySQL: database
- Agile: software development methodology

We chose this design because it offered us familiar frameworks and libraries to work with, as well as a good option for a data visual project. On top of that, working in an agile development helped the software group work with continuous improvement, and iterations that allowed us to develop at a rate where we would still be able to integrate our work with the power side.

The user interface was designed to be very simple and easy to use. We knew that we wanted current/instantaneous readings to be displayed on the main page, then branch out with buttons to separate dialogs for other functions like instant history, history, leveling calibration, and leveling adjustment.

Figure 13: Main screen, showing current readings

Figure 13 is the default screen that pops up upon search. This screen is meant to show current readings that update upon refresh. There are many different buttons on this screen, each corresponding to the row it lies on. Current and voltage readings have both instant history and history buttons due to the fact that these are non-calculated values that are being read from the RV. Power is being calculated from the corresponding AC or DC voltage and current readings by multiplying them together. Outside of AC and DC, there is the leveling row that has two different buttons, one for calibration, and one for the calculated adjustment data after the said calibration button has been pressed. This was designed this way to make sure there are not any unnecessary calibrations every time you want to see the adjustment, if there was only one button. The calibration button is meant to be pressed rarely only when you, obviously, need to calibrate the leveling. The following buttons are labeled above, and can be seen in different photos below:

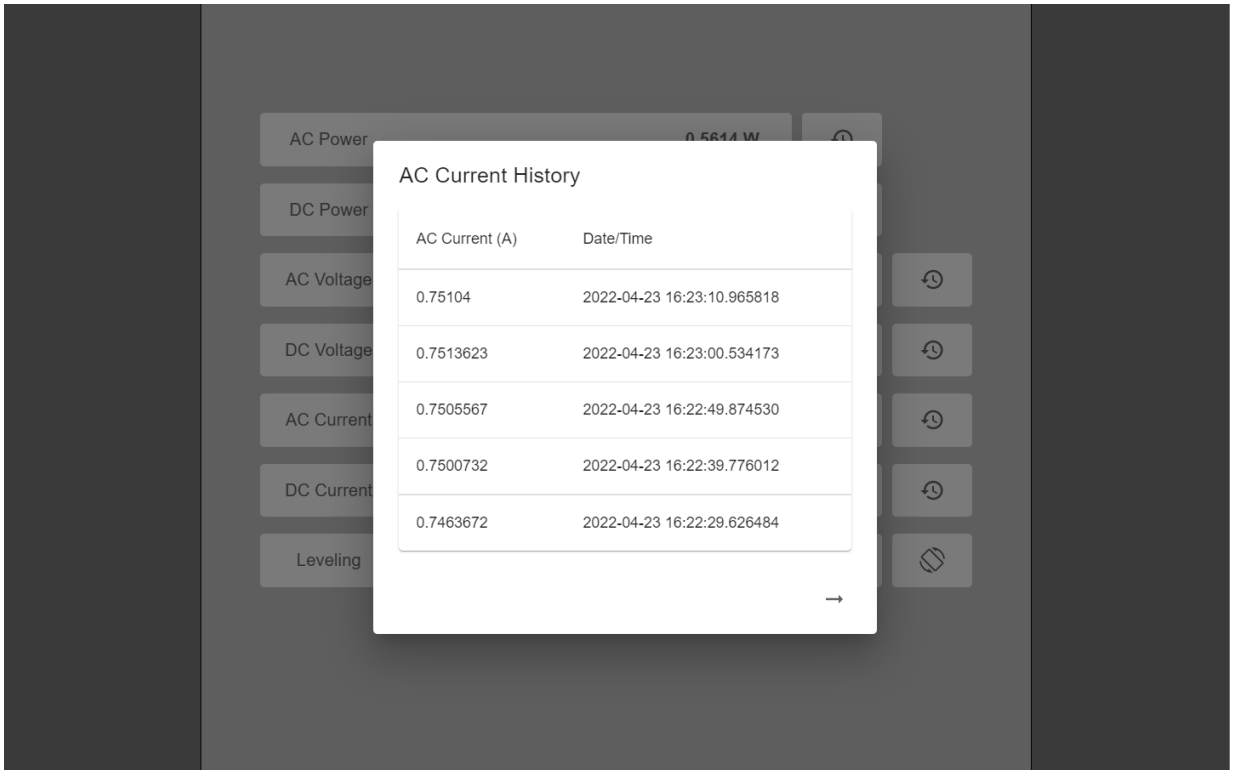

Figure 14: History screen (1)

Figure 14 shows the history screen, which can show either AC or DC power, voltage, or current data history, depending on the button clicked. In this photo, it shows only a history of 5 readings, which was used only for testing maximum storage conditionals, and cleaning purposes. It also is taking in readings every 5 seconds, but as planned for RV can be changed to 60 seconds (every minute). Like mentioned, there is a cleaning mechanism that checks if it has reached a manual input ceiling amount of data, that will delete the oldest data and allow a new one to be entered into the table.

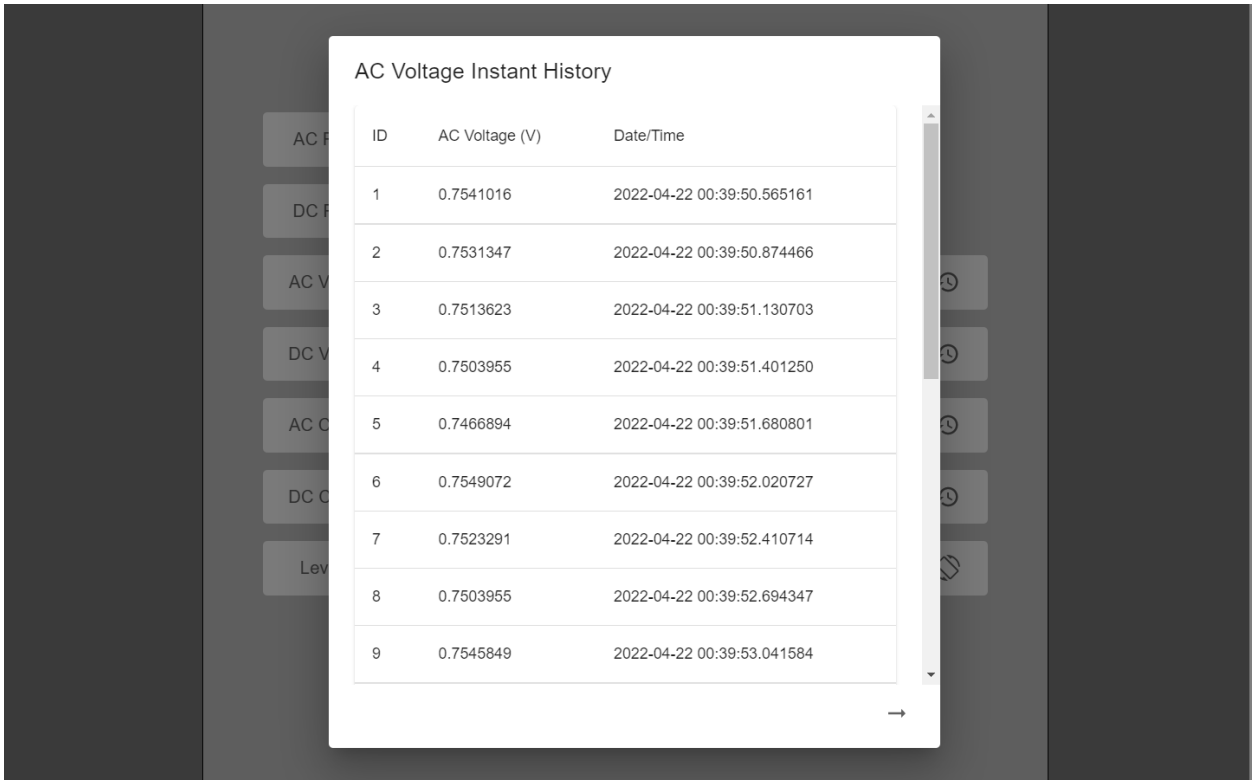

Figure 15: Instant history screen (2)

Figure 15 shows the instant history screen. This gives you a maximum of 20 readings, being read every second (may take up to 20 seconds to render). The design was to make this into a waveform, but as of now, is showing the instantaneous data within a table. We decided to include this in our design to not only act as a placeholder for a possible future waveform generation, but in general to give the RV user an idea of the incoming data in very recent history (less than a minute).

The calibration button (3) is simply a button that is pressed when you want to calibrate the leveling. This basically will run a script to calculate the front/back and side/side average for leveling the RV using an IMU unit. Upon clicking, it will determine whether or not the calibration was successful, and give you a message depending on the outcome. Once this is completed, the adjustment button (4) will contain data after a small amount of time.

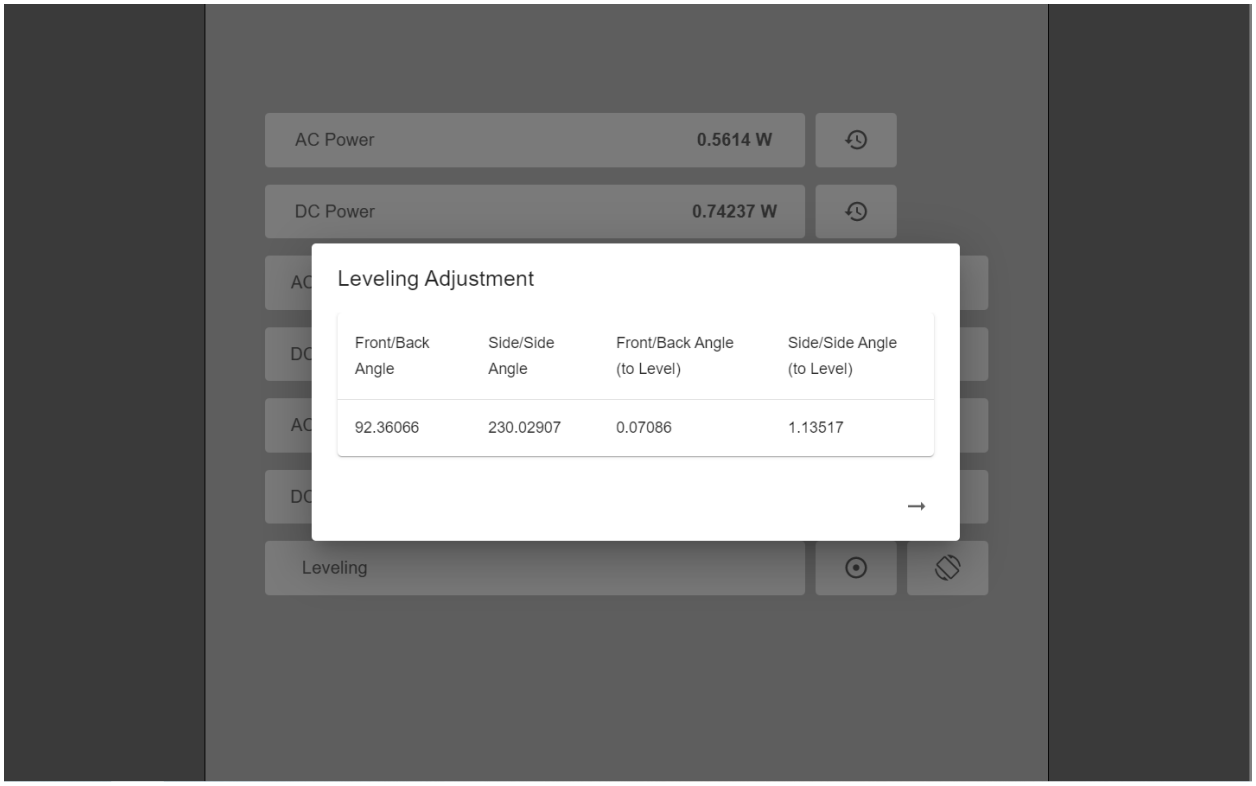

Figure 16: Leveling adjustment screen (4)

Figure 16 is the leveling adjustment screen. This shows the user the updated front/back and side/side angles along with the “toLevel” angles. This is done by taking accelerometer data in m/s^2 and translating it to an angle, using an atan2 function.This tells the user how to level their RV with ease. Like mentioned before, this will show data, as long as the RV has been calibrated before, with the calibration button (3).

Design Testing

Power Sensors

Testing for the power systems portion of the project was divided into 4 different phases, one for each specific sensor. This consisted of testing the DC current and voltage, as well as the AC current and voltage sensors.

1. **DC Voltage**: Testing for the DC voltage sensor was the most straightforward, as we were able to use standard bench power supplies and measurement tools available to us to replicate the 12V DC input signal.
2. **DC Current**: We were unable to find a safe way to test the DC current monitoring circuit. This is due to the fact that we did not have the necessary tools to simulate a high current DC load. We expected that the DC current could be several amps, which is much higher than we could feasibly simulate in a laboratory setting.
3. **AC Voltage**: Testing for the AC voltage was not conducted due to timing limitations. However our plan for testing will be discussed in the AC current portion of the design testing.

4. **AC Current**: Testing for the AC current monitoring was conducted on a 90W box fan. We conducted AC current testing on a relatively low current box fan because it was necessary to use an AC current device that was disposable as we needed to splice into its insulation to install the current transformer. We installed the current transformer in the device and tested the fan at varying speeds to ensure that our current monitoring circuit was functioning correctly under various conditions. It was also our intention to test our AC voltage circuit by taking advantage of where we spliced into the box fan. Our plan was to plug our AC voltage sensor into the place where we spliced into the wiring of the box fan, so we could read 120V coming from the outlet. Due to timing constraints we were unable to complete the AC voltage testing.

Embedded Systems

Testing for the embedded systems portion of the project boiled down to two aspects: data handling and power efficiency. Data handling testing involved the speed and accuracy at which ADC readings could be acquired and transmitted to the server. Power efficiency testing involved finding ways to reduce power consumption of the Raspberry Pi Pico to prevent excessive power draw.

The first data handling test consisted of the system's ability to gather accurate data. First, the system was calibrated against a DC voltage source, and ADC measurements were measured and plotted. It was found that the ADC had no observable offset voltage, and had a linear slope across 100mV input steps from 0V to 3.3V. This proved sufficient for our application, and was then tested against the output of the DC voltage sensor. A known voltage was applied to the DC power sensor, and the output voltage was measured with both a multimeter and the ADC. It was found that there was a 3mV deviation between the multimeter and ADC measurements during operation. As the measurement which requires the most stringent accuracy, the DC voltage measurement, must be within 5mV, this deviation proves acceptable for our application .

The next data handling test consisted of ensuring that the system could sample at a speed which could accurately capture AC sensor data. As the AC power used in the system would be 60Hz, the output of the AC power sensor would also be 60Hz. To sufficiently sample the AC power sensor output, the Raspberry Pi Pico would have to sample the output at 120Hz per the Nyquist theorem.However, it was desired to oversample the output wave in an attempt to capture the peak output value of a given AC power sensor. If the peak output of the power sensor is not captured in a given data acquisition sequence, it will not be representative of peak power draw. Therefore, it was determined that the ADC should oversample at a rate over 120Hz to capture peak sensor outputs. However, sampling would have to occur across two different channels which shared the same ADC, which introduces additional bandwidth limitations.

To test the sampling rate, a 60Hz sine wave was applied to two channels of the ADC with different amplitudes via a waveform generator. The Raspberry Pi Pico clock speed was increased to 250MHz, and the ADC was sampled as rapidly as possible until the 2 kilobyte transmission buffer was filled. The data was then transmitted to a script which plotted measured data for simple visualization during testing. One test with one channel reading a 1V sine wave centered around 1V and the other channel reading a 1.5V sine wave centered around 1V is shown below in **Figure 17**. All peaks met the $10mV_{RMS}$ threshold for our application across

multiple tests. In decoding on the server end, these peaks can be found using a maximum or minimum value search function. It is shown in **Figure 17** that the Raspberry Pi Pico can capture five samples per cycle per channel, giving the ADC sampling rate of roughly 600Hz. This gives each channel of the ADC in a two-channel setup a sampling rate of roughly 300Hz. While the waveform is not completely captured, the peak values are captured, which is necessary to calculate power consumption.

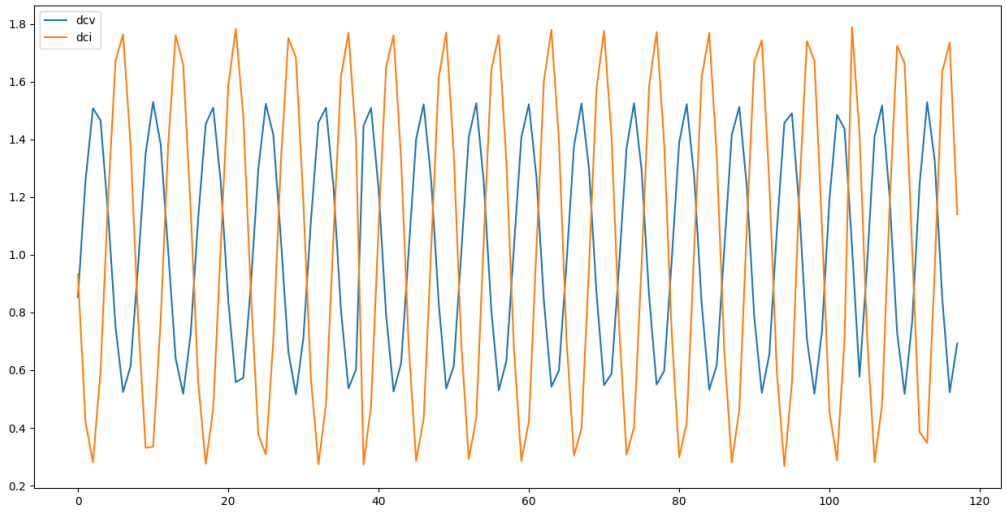

Figure 17: Data Acquisition Test with 60Hz Input Sinusoids of Various Amplitudes

The last data handling test considered the total amount of time required to acquire ADC readings and transmit them to the server. On the Raspberry Pi Pico, approximately 500 milliseconds are required for overhead to accelerate the clock to 250MHz without causing any clock issues and data loss. Additionally, with the given sampling frequency compared to the measured wave, it is estimated that it takes roughly 250 to 300 milliseconds to collect 2 kilobytes of ADC readings. When combined with the estimated 200 milliseconds of overhead necessary to send the data via TCP socket, it takes roughly one second from measurement request by the server to measurement reception on the server. While not instantaneous, the one second timeframe was the minimum which could be achieved with the given setup. Lastly, packet loss was never observed over various testing intervals ranging from a few minutes to one hour, so it can be assumed that there is no packet loss in operation.

As a power monitoring system which acts as a significant load on the system which it is monitoring is useless, a method of power savings was desired. It was found during research that power consumption should scale linearly with clock speed, with higher clock speed correlating to increased power consumption and vice versa. These differences were noticeably large; reducing clock speed from 150MHz to 16MHz was given by the datasheet to reduce power consumption by ~80%. Therefore, the primary method of power savings for the Raspberry Pi Pico would involve increasing the clock speed during data handling, and reducing the clock speed during idling.

To test this theory, a clock speed adjustment method was implemented. During setup and data handling, the clock speed was increased to 250MHz via the python function machine.freq() to ensure sufficient sampling and system responsiveness. While waiting for a measurement request or otherwise idling, the clock speed was reduced to 16MHz. Using linear approximations based on datasheet current consumption values, it was estimated that the microcontroller will consume a maximum of 60mA of current while the clock speed is adjusted to 250MHz and 7mA of current while the clock speed is adjusted to 16MHz. If a measurement is queried once per minute, and the high clock speed data handling takes one second, the microcontroller's average current draw will be 7.883mA. This gives an average power consumption of 26.015mW at the Raspberry Pi Pico's operating voltage of 3.3V, assuming power loss is negligible.

In testing, current draw was measured using a multimeter during different modes of operation. It was found that the Raspberry Pi Pico consumed 36.26mA of current during setup, 44.2mA of current during data acquisition and transmission, and 7.59mA of current while idling. Setup power consumption can be neglected as it represents a minimal fraction of system load during long term operation. With setup power consumption being neglected and a measurement being queried once per minute, the microcontroller will draw an average current of 8.2mA while running during long term operation. At the Raspberry Pi Pico's operating voltage of 3.3V, this gives an average power consumption of 27.06mW. If the system clock speed is left at 250MHz during operation, the system would consume roughly 145.86mW, representing a ~539% increase in power consumption. During operation, this idle feature will help to conserve power consumed by the Raspberry Pi Pico and ensure that the power monitoring system does not become a parasitic load on the system which it is tasked with monitoring.

Testing of the IMU was relatively simple; all that was necessary was a calibration method and accuracy check. All testing was conducted while the IMU was fixed within a 3D-printed enclosure to simulate use in the RV.

As calibration was achieved by collecting a large amount of samples and averaging them, it was necessary to know the number of samples needed to be guaranteed an accurate calibration. To test this, a quantity of sample points were arbitrarily chosen, and the calibration script was run while the IMU and enclosure were fixed in one place. The calibration script was run ten times, and the deviation in calibrated values was noted. Calibration sample points were increased until the change in calibrated angle values between separate runs was less than 0.1 degrees. Using this testing method, it was found that 100 samples was the minimum number of calibration sample points necessary to guarantee less than 0.1 degrees of deviation between separate calibration runs. Therefore, calibration is performed by averaging 100 sample points.

To test the accuracy of the IMU, a calibration was performed with the IMU and enclosure orientation relative to a protractor. Once calibration was completed, the IMU was rotated in either the front-to-back or side-to-side angles, and the angle of the IMU and enclosure was measured via the protractor. Then, the IMU angle was found via a test script running on the Raspberry Pi Zero W which returned both the front-to-back and side-to-side angles. The angle measured via the protractor and the angle measured via the IMU were then compared for accuracy. While the protractor was only accurate to 1 degree, which is less than the specified +/- 0.5 degrees specified for the project, it was found that when the IMU was oriented to the nearest degree on the protractor, the test script returned an angle which was within +/- 0.1

degrees of the protractor, which correlates with the 0.1 degree deviation from calibration and error associated with being accurate to the nearest degree. There was roughly +/- 0.05 degrees of noise in the IMU measurement, but this is small enough that it can be ignored. Accuracy beyond 0.1 degrees is not necessary for an end user, and will be rounded off at the user interface. This testing, along with accuracy guarantees made in the ICM20948 datasheet, proved that the IMU would be sufficient for our application.

Data Handling

In order to test the data handling aspect of our project, we need to do some sort of integration testing between the different components that are communicating with each other. This would include both the data transfer from pico to zero and zero to server.

First, the data transfer from pico to zero. This mainly consisted of AD-HOC tests to test what type of incoming data there was, how many intervals between readings there were, and when to capture/collect data at an efficient rate that we would need for this web application. Since we were working with two separate picos (for AC and DC readings), we tested both picos by writing python scripts, opening up a serial port for both picos using two separate UART drivers. Within these, we were able to complete our random tests to see whether or not we needed to request data from the picos, or if we could contain a constant stream of incoming data, and parse it at our time of need. We tested it and found that we were capable of not needing a requesting based approach, rather we could read a line of incoming data every ‘x’ interval. Other tests included trying different timeout and sleep ranges between cycles to make sure our web application will run smoothly and not get backed up, and testing to see if there was inconsistency within the incoming data by running it at different times/lengths. In terms of using “real” data for testing, we were able to hook up to a generator at a lab bench. We were able to monitor and parse through this “real” data that we would be able to manually enter and mimic the RV output.

Second, the data transfer from zero to server. This part of our testing was to see if there was any inconsistency or delay with having multiple get and post requests happening within a second of time. This would include having different timeout lengths, querying the same table to get, set, and empty table data all within a very small amount of time, and doing random button clicking to see if there are any random delays or incomplete data transfers between clicks.

User Interface

Overall testing for the user interface was done via unit testing using a test framework known as Jest. A test suite was created that would model and test various aspects of the web application for proper function, such as, but not limited to, button press, page refresh, and various aspects of page interactivity. Integration testing was also done to ensure that data values from the various sensors were being transmitted to the front-end and also stored for future processing and/or retrieval in the backend servers. From this, test cases were also created to not only examine and verify the functionality of data retrieval from the backend servers, but also ensure that the data values were correct and uncorrupted. We also made use of heavy manual testing and visual inspection to verify that the web application loaded correctly, clickable events happened as they were meant to, and that the application was easy to use from a user experience perspective.

Implementation

Due to slowed progress in development of the power sensors, the physical implementation of the system was limited. However, there were still components which could be physically implemented within the RV at the end of the development cycle.

The IMU and Raspberry Pi Zero W were implemented within the 3D printed enclosure as described in **Design Overview: Embedded Systems**. PLA material was deemed adequate for this enclosure, as it would not experience stress capable of damaging the enclosure when mounted within the RV cabin during normal operation.The enclosure consisted of mounting points for device PCBs placed at dimensions specified by mechanical drawings and fabrication prints with additional space for wires to be routed as needed. Within the design, 2.5mm bolts were used to fix the device PCBs to their respective mounting holes. **Figure 18** shows the IMU and Raspberry Pi Zero W mounted within the enclosure.

Figure 18: Raspberry Pi Zero W and IMU Mounted Within Enclosure

Enclosures were not created for the Raspberry Pi Pico and ESP8266 for either the AC or DC power sensor. As designs for these sensors were not finalized, many considerations could not be made for the enclosures for these devices. These considerations include enclosure material, physical size constraints, proximity to the associated sensors, shielding from possible wireless noise and interference, mitigation of mechanical stress during operation, protection

from environmental contaminants, and potential heat dissipation needs. While a vented or air-cooled design was considered, the openings needed to achieve sufficient air flow would likely necessitate additional filtering to prevent the introduction of environmental contaminants. Shock and stress can be easily mitigated using low-cost polyurethane bushings, but space requirements could prevent their use depending on the necessary size and/or density. Lastly, as there is likely to be other devices which introduce interference and noise in sensor readings, it may be desired to shield the enclosure, However, shielding may reduce the effective data transmission range of the ESP8266, causing data corruption or loss during operation. While many considerations were made in the design of these enclosures, none were completed due to the lack of finalized AC and DC sensor designs.

Appendix

User Guide

Physical Sensor Installation

AC Sensor Installation

NOTE: This physical design for the AC circuit is unfinished by the plan for installation is below:

1. Mount the Power Outlet Box to the back of the RV within reasonable distance of the shore power plug
2. Plug the male NEMA 14-50P connector from the RV into the 125/250V RV Power Outlet Box containing the AC power sensors
3. Connect the male NEMA 14-50P connector from the Power Outlet Box containing the sensors into the Outlet Box fixed onto the RV where the connection to shore power is

DC Sensor Installation

NOTE: We were unable to finish a design for DC sensor installation, but the expectation was to have a fixture similar to the AC sensor installation, where we created a fixture that would act as a middleman between the batteries and the RV.

Sensor Connection To Server

NOTE: For proper operation, a router or other access point must be used on the RV. The sensor must wirelessly connect to a router in order to form a connection with the server, and will not be able to send data without a router. The SSID and password for this router should be known for setup.

1. If this is NOT first boot of the system, skip to **Step 7**
2. The server should already be connected to wifi by default. If not, run “raspi-config” within a terminal on the server and complete the wi-fi setup instructions. Identify the IP address of the server by running “ifconfig” within a terminal. Make note of the IP address for **Step 5**.
3. Connect the Raspberry Pi Pico to a computer via USB and enter the main.py file using the Thonny IDE.

4. In the espSetup() function, there should be two variables which are named “wifiSSD” and “wifiPWD”. Change these variable values to the router SSID and router password, respectively.
   NOTE: For proper function, the router must have a password. The system will not work if the password is left blank.
5. In the espTCPSetup() function, there should be one variable which is named “server”. Change this variable value to the server IP address found in **Step 2**. DO NOT EDIT THE VARIABLE NAMED “port”.
6. Once edits are made, save the file to the Raspberry Pi Pico and unplug the Pico from the computer.
7. Activate power to the Raspberry Pi Pico and ESP8266 wi-fi coprocessor. The microcontrollers will begin running setup scripts to connect to the router and TCP server.
8. The Raspberry Pi Pico’s onboard LED will indicate when the system is connected to the router. If power sensor readings are visible on the user interface, then the TCP connection is good.
9. If any debugging is needed, it can be accessed by connecting to the Raspberry Pi Pico via USB on a computer and utilizing the “Shell” feature within the Thonny IDE to get relevant debugging information.

Interaction with User Interface

Leveling Calibration:

NOTE: Leveling calibration should be performed after the system is first installed, once every 6 to 8 weeks during use, or after a period of extended storage.

1. Perform RV leveling manually using a bubble level or other leveling method.
   NOTE: DO NOT USE THE LEVELING INTERFACE FOR THIS STEP. As the goal of this step is to calibrate the leveling interface, this may contribute inaccuracies to the calibration. A leveling method independent from the leveling interface will minimize calibration errors.
2. Within the web interface, navigate to the “Calibrate” button. Check that the RV is level before continuing.
3. Press the “Calibrate” button within the web interface once ensuring the RV is level. The system will then take a large amount of measurements.
4. The system should complete calibration within ~5 seconds. Now, the leveling interface can be utilized for RV leveling.

Obtaining a Measurement Quantity:

1. Enter the user interface via the website name assigned to the server on a web browser.
2. The leftmost fields indicate the quantity which is being measured. To the right of those identifiers is the most recent measurement of that quantity.
3. There are two buttons to the right of the most recent measurement which can be selected. The first button shows waveform data, and the second button shows historical data

Previous Designs / Changes from 491

- Functional Requirements: Removal of temperature measurement

In the previous semester, one functional requirement involved the monitoring of the RV cabin temperature. This temperature measurement was to be viewable on the user interface and accurate to 1°F.

However, this temperature measurement was removed from the final list of functional requirements. It was seen as a feature which would be implemented last, as it would likely be redundant due to an onboard thermostat which also provided cabin temperature. Additionally, calibration would be difficult due to an inability to track temperature sensor variations and drift over time. Most available parts are only accurate to 1°C, or ~2°F, which is not accurate to our specified functional requirement. Therefore, cabin temperature monitoring was left off the list of functional requirements.

- Embedded Systems: Updating from USB to wi-fi data transmission

A previous design utilized USB data transmission between the Raspberry Pi Zero W and Raspberry Pi Pico. A reading could be queried and a response could be transmitted via serial communications. In testing, this provided both accurate data resolution and sufficient transmission speed. However, implementation presented issues which would make this design not viable.

The first issue required drilling holes through the walls of the RV to run a USB cord from the Raspberry Pi Zero W to Raspberry Pi Pico, which could be undesirable to some users. The second issue involves the physical length of the USB cord. It was likely that 20+ feet of USB cord would need to be used to connect the Raspberry Pi Zero W to a given Raspberry Pi Pico. USB standards specify that USB cords not exceed 5 meters, which is less than the 20+ feet necessary for implementation. Violation of USB protocol meant the system could behave unreliably, slowing data transmission or even losing data packets. The third and final issue involved the limited USB ports on the Raspberry Pi Zero W. As the Raspberry Pi Zero W only has one USB port, a USB port extender was necessary for more than one device. The USB port extender presented some issues in use and added an unattractive bulk to the final implementation.

To avoid the above issues, the USB data transmission method was overhauled to the current system which utilizes an ESP8266 wi-fi coprocessor to transmit data between the Raspberry Pi Pico and Raspberry Pi Zero W. It was found in testing that the overhaul was successful, as data could still be transmitted at both an accurate resolution and sufficient speed.

Other Notes and Issues

- IMU Addressing Issues

As the I2C protocol relies on all target devices to have an address, knowledge of the IMU I2C address was necessary to write the leveling code. The address provided by Adafruit Industries was 0x69, but when searching for I2C devices at that address on the Raspberry Pi Zero W, errors were encountered. However, there was a device at the address 0x68 which appeared to be the IMU. After further testing, it was found that the IMU's I2C address was 0x68 after running some test code to obtain various acceleration, gyroscope, and magnetometer measurements. Therefore, we assumed the IMU had an incorrectly assigned address.

There was an onboard contact point which could be bridged with solder to change the device address from 0x69 to 0x68, but it was not initially bridged. One test consisted of bridging the contact point to observe if the address changed, which proved unsuccessful; bridging the contact point with solder did not change the address. Therefore, we believe the IMU had likely been assigned an incorrect address during some step in the fabrication process. All code associated with the IMU was changed to utilize the device's observed address and not the datasheet listed address.

- 16-bit ADC reading issues

While the onboard ADC for the Raspberry Pi Pico is 12 bits, the builtin micropython function for collecting ADC readings returns a 16 bit number. This introduced an issue, as it was unknown where the extra 4 bits within the readings were originating. Additionally, it added some noise to the ADC reading, which could lead to small miscalculations. In testing, it was found that the four least significant bits of the 16 bit ADC reading were mostly noise. This was determined by applying a DC voltage to the ADC and observing the measured output.

To mitigate this issue, the 16 bit ADC reading was bit shifted such that the four least significant bits of the reading were removed. This returned a 12 bit ADC reading which experienced a reduction of noise while still retaining 1mV accuracy. The bit shifting solution was implemented in the final design and proved to be an adequate fix while introducing little additional code overhead.

- Part Stock Issues

As noted in the above sections, some design decisions were made based on the availability of parts. While our design does not utilize many components which were affected by the silicon shortage experienced during 2021 and 2022, some portions of development were slowed when part stock issues arose. Design decisions related to part stock issues are discussed below.

While the ESP8266 wi-fi coprocessor is currently NRND (not recommended for new designs) per its manufacturer, they were purchased before the NRND lifecycle phase by a group member who utilized them for homemade wi-fi development boards. The new recommended part is the ESP32, which features Bluetooth and other connectivity options, but is only in stock periodically through trusted distributors. Rather than waiting for the ESP32 to become available and creating a development board to interface with the hardware, the ESP8266 was utilized as it was readily available and has only recently been placed in the NRND lifecycle phase.

Raspberry Pi devices faced part stock issues during the latter half of the course. Early in the development phase, a Raspberry Pi Zero W was purchased for use within the project. However, the Raspberry Pi Zero W did not come with headers pre-installed, and the IMU required headers for its hardware interface. An unknown error occurred during the header installation process, which destroyed the Raspberry Pi Zero W. At the time it was destroyed, there were no Raspberry Pi Zero Ws available; the only available Raspberry Pi devices were the Pico microcontroller, which did not have the processing power required, and the Raspberry Pi Compute Module, which consumed too much power during operation and required an additional carrier board for hardware interfaces. Therefore, one group member volunteered their Raspberry Pi Zero W for the remainder of the project, which was returned at the end of development and testing. All relevant software was pushed to the group github repository since the device was returned to the group member.

- Use of UART interface for final demonstration

As the discovery that the design violated USB protocol standards (USB cable over 20 feet) occurred very late in the development process, there was little time left to integrate the new TCP socket based query system to the final design. Therefore, the TCP socket based query system was left as a proof of concept, and all demonstrations were done using the UART interface. Demonstrations could occur without violating USB standards, so the system remained.

The UART interface essentially accomplishes the same end goal of transmitting data from Raspberry Pi Pico to Raspberry Pi Zero W, but has one major drawback: power consumption. While the TCP socket based query system only returns data when queried and idles all other times, the UART interface provides a constant stream of data which is read in set intervals. This means that with the UART interface, the Raspberry Pi Pico is constantly running at a maximum clock speed, drawing a significantly larger amount of power. If more time or resources were allocated to the transition to the TCP socket based query system, or if the discovery of the violation of USB standards were found earlier in the development process, the integration could have been completed for demonstrations. However, as mentioned above, time restrictions prevented the full integration of the TCP socket based query system, leaving only the UART interface for demonstrations.

Embedded Code

main.py for TCP socket based query system

```
#main.py
#sends ADC data over TCP connection to server
#Written by: Nick Moser
#Last edited: April 17, 2022

from machine import UART, Pin
import utime
import sys

#used to set up ESP device
#uses espWrite function to run through setup command list
def espSetup(espDevice,led):
    #esp station mode
    cwmode = '1'
    #wifi SSID and password
    wifiSSID = '"testSSID"'
    wifiPWD = '"testPWD"'
    #list of setup phrases to use
    setup_phrases =
['AT+GMR','AT+CIPSTAMAC?','AT+CWMODE='+cwmode,'AT+CWJAP='+wifiSSID+','+w
ifiPWD]
```

```
    #run through setup phrases
    for i in range(len(setup_phrases)):
        #write setup phrase and get write status
        espStatus = espWrite(espDevice,str(setup_phrases[i]))
        #if the write failed, return failure and give a response
        if(espStatus != 1):
            print("no ESP device found. Please connect one to UART")
            return 0
        #wait a quarter second between writes. ESP cant handle full speed
        utime.sleep(0.25)
    #set the LED high if setup and wifi connection is complete
    led.high()
    #return one if successful
    return 1

#code to simplify writing to ESP device
#takes uart object and write phrase as arguments
def espWrite(espDevice,phrase):
    #append newline to any written phrase
    endcode = '\r\n'
    #write the phrase and endcode
    espDevice.write(str(phrase)+endcode)
    #wait a quarter second. ESP cant handle full speed
    #if there is some returned ESP info associated with the write
    while espDevice.any() == 0:
        continue
    while espDevice.any():
        #capture ESP response
        espResponse = str(espDevice.readline())
        #next two lines parse for printing to serial monitor
        espEndIndex = len(espResponse)
        print(espResponse[2:espEndIndex-5])
        #if the written phrase contains the wifi connection command
        if(espResponse.find('CLOSED') != -1):
            while espDevice.any() == 0:
                continue
        if(phrase.find('CWJAP') != -1):
            #if the response is that the IP is acquired
            if(espResponse.find('OK') != -1):
                #return success
```

```
            utime.sleep(1)
        #if the response is any other connection phrase
        else:
            #wait until something appears in the buffer
            #which will reset the while loop
            while espDevice.any() == 0:
                continue
    if phrase.find('CIPSTART') != -1:
        if espResponse.find('OK') != -1:
            break
        if espResponse.find('ERROR') != -1:
            break
        else:
            while espDevice.any() == 0:
                continue
    if espResponse.find('OK') != -1:
        break
if espResponse.find('OK') != -1:
    return 1
else:
    return 0

#set up tcp connection
def espTCPSetup(espDevice):
    #specify protocol
    protocol = '"TCP"'
    #specify server to be connected to
    server = '"testServer.local"'
    #specify port to connect to at server
    port = '50010'
    #create conenction command
    connStart = 'AT+CIPSTART='+protocol+','+server+','+port
    #push connection command
    if espWrite(espDevice,connStart) == 0:
        return 0
    else:
        return 1

#write to a tcp connection
def espTCPWrite(espDevice,data,dataLen):
```

```
    endcode = '\r\n'
    #create write command
    dataSend = 'AT+CIPSEND='+str(dataLen)
    #push write command
    espDevice.write(dataSend+endcode)
    #wait for esp response
    while espDevice.any() == 0:
        continue
    #when esp responds
    while espDevice.any():
        #get & print response
        espResponse = str(espDevice.readline())
        espEndIndex = len(espResponse)
        print(espResponse[2:espEndIndex-5])
        #if there was an error in transmission, exit
        if espResponse.find('ERROR') != -1:
            print("cipsend failed")
            return 0
        #if esp is anticipating data to be sent
        if espResponse.find('>') != -1:
            #send data
            espDevice.write(str(data))
            #wait for response
            while espDevice.any() == 0:
                continue
        #if data has been transmitted, leave command
        if espResponse.find('SEND OK') != -1:
            break
        else:
            while espDevice.any() == 0:
                continue
    return 1


#PROGRAM STARTS HERE

machine.freq(250000000)
#Create UART object for ESP device
espDevice = UART(0, baudrate=115200, tx=Pin(0), rx = Pin(1), timeout = 2)
#setup information for ADC
```

```
adc_pins = [26, 27]
adc_tags = ['dcv','dci']
adc = []
#create ADC objects
for i in range(len(adc_pins)):
    adc.append(machine.ADC(adc_pins[i]))
#information for sending tcp data
sentBytes = 0
dataSend = ''

#Create LED object to indicate connection to wifi
led = Pin(25, Pin.OUT)

#if startup command fails
if espSetup(espDevice,led) != 1:
    #indicate the startup failed, exit the program
    print("Setup failed")
    sys.exit()

#if tcp connection fails
if espTCPSetup(espDevice) != 1:
    print("TCP connection failed")
    sys.exit()

machine.freq(16000000)

while True:
    #if nothing has been queried from tcp connection
    while espDevice.any() == 0:
        continue
    #if tcp connection requests something
    else:
        espResponse = str(espDevice.readline())
        #if tcp connection wants a measurement
        if espResponse.find('meas') != -1:
            utime.sleep(0.25)
            machine.freq(250000000)
            utime.sleep(0.25)
            #while buffer has less than 2kb of data
            while sentBytes < 2024:
```

```
            #collect ADC measurements from both channels
            #keep track of data in buffer
            for i in range(len(adc)):
                adcReading =adc[i].read_u16()>>4
                tagData = adc_tags[i]+'?'+str(adcReading)+';'
                dataSend+=str(tagData)
                sentBytes += len(tagData)
        #once buffer has filled, append end tag
        dataSend+=str('end')
        sentBytes += 3
        #write tcp data to server
        if espTCPWrite(espDevice,dataSend,sentBytes) != 1:
            print("write failed")
        #reset buffer and bytes in buffer
        dataSend = ''
        sentBytes = 0
        machine.freq(16000000)
        utime.sleep(1)
```

Old main.py for demonstration with UART interface

```
#new update
#increases clock frequency for better sampling
#converts reading to actual voltage
#Nickolas Moser
#Last Edited: April 7, 2022

#import necessary modules
from machine import Pin,ADC,UART
import utime

#set up ADC pins
def ADC_setup(adc_pins):
    adc_channel = []
    for i in range(len(adc_pins)):
        adc_channel.append(i)
        adc_channel[i] = machine.ADC(adc_pins[i])
    return adc_channel

#set up serial communications
def serial_setup(led):
```

```
    led.toggle()
    serial_object = machine.UART(0, 115200)
    utime.sleep(1)
    led.toggle()
    utime.sleep(1)
    return serial_object

def uart_handler(RPi_zero,led):
    led.toggle()
    while RPi_zero.any():
        print(RPi_zero.read(1))
    utime.sleep(1)
    led.toggle()

#get an adc reading
def get_reading(channel):
    window_size = 5
    samples = []
    samples_sum = 0
    for i in range(window_size):
        samples.append(3.3*((channel.read_u16() >> 4)/(2**12)))
        samples_sum = samples_sum + samples[i]
    reading = samples_sum/window_size
    return reading

#program starts here

#make it work a bit faster
machine.freq(250000000)

#global variables
adc_pins = [26,27]
adc_quantity = ['dcv','dci']

#device object setup functions
led = Pin(25, Pin.OUT)
led.toggle()
utime.sleep(1)
#RPi_zero = serial_setup(led)
adc = ADC_setup(adc_pins)
```

```
led.toggle()

#while device is turned on
while True:
    for i in range(len(adc)):
        print(adc_quantity[i],"?", get_reading(adc[i]))
```

serial_listener.py for testing of UART interface

```
import sys                                          #needed to do system
level stuff
import glob                                         #needed to list available
/dev/ttyA devices
import serial                                       #needed to communicate
over serial
from time import sleep                                      #not needed
outside testing
import matplotlib.pyplot as plotter

def adc_parse(data_in):                             #function to parse serial
ADC data
        data_in_str = str(data_in)                  #convert input to string
        data_in_rm = data_in_str.split('\'')        #remove b' at beginning of
string
        clean_data = data_in_rm[1].split('\\r')     #remove \\r\\n at end of
string
        if clean_data[0].count('.') > 1:            #if it accidentally picks up
two readings
                return
        elif clean_data[0].count('.') < 1:          #if it accidentally picks up
no readings
                return
        adc_reading = clean_data[0].split('?')
        return adc_reading

def acquire_picos():                                #function to connect to
pico devices
        devices = glob.glob('/dev/tty[A]*')         #get available /dev/ttyA
devices
        open_devices = []                           #create return list
```

```
        print("Picos available: ", len(devices))                    #print available /dev/ttyA
devices
        if len(devices) == 0:                                       #exit if no picos are
available
                print("No picos available. Now exiting")
                exit()
        for i in range(len(devices)):
                open_devices.append(i)                                  #add to list of
available devices
                open_devices[i] = serial.Serial(devices[i],115200)      #open device for
communication at baud rate of 115200
        print("Picos opened: ", len(open_devices))                      #print picos
connected in this step
        return open_devices                                             #return opened
devices

                                                                    #program starts HERE
                                                                    #below are necessary
global variables
open_devices = []                                                   #list of opened serial pico
connections
device_data = []                                                    #list of device data
collected from picos
collected_data = []
dcv_reading = []
dcv_datapoint = []
dci_reading = []
dci_datapoint = []

open_devices = acquire_picos()                                          #open all available
/dev/ttyA devices
for i in range(len(open_devices)):                                  #create device data list
        device_data.append(i)

print("devices acquired")

#for data acquisition testing
for i in range(500):
        open_devices[0].flush()
        reading = adc_parse(open_devices[0].readline())
```

```
        if reading[0] == 'dcv ':
                dcv_reading.append(float(reading[1]))
                dcv_datapoint.append(len(dcv_reading))
                print(len(dcv_reading))
        elif reading[0] == 'dci ':
                dci_reading.append(float(reading[1]))
                dci_datapoint.append(len(dci_reading))
        else:
                continue
plotter.plot(dcv_datapoint,dcv_reading, label = 'dcv')
plotter.plot(dci_datapoint,dci_reading, label = 'dci')
plotter.legend()
plotter.show()

while True:
        for i in range(len(open_devices)):                      #run through list of opened
devices
                device_data[i] = adc_parse(open_devices[i].readline())          #read from
serial port
                open_devices[i].flush()                                 #clear buffer once
reading has been collected
                print(device_data[i][0],float(device_data[i][1]))
                sleep(0.1)                                      #sleep, time is quicker
than pico sleep for testing purposes
```

socketListener.py to test TCP socket based query system

```
import socket                           #needed for socket comm
import matplotlib.pyplot as plotter             #plotting for testing data
from time import sleep                  #sleep between intervals

def dataParser(recvData):                       #function to parse returned data
   dcv = []                             #arrays for measurements
   dci = []
   totalBits = 2**12
   splitData = recvData.split(';')              #split at all semicolons
   for i in range(len(splitData)):              #run thru new split-up array
      taggedData = splitData[i].split('?')          #split all indices at ?
      if taggedData[0].find('dcv') != -1:           #if tagged as dc voltage
         dcv.append(3.3*(float(taggedData[1])/totalBits))#add data to dc voltage
      if taggedData[0].find('dci') != -1:           #if tagged as dc current
```

```
            dci.append(3.3*(float(taggedData[1])/totalBits))#add data to dc durrent
        if taggedData[0].find('end') != -1:             #if end tag is found
            break                                   #exit loop
    return dcv, dci                                 #return dc voltage and current

#PROGRAM STARTS HERE

dcv_points = []                                     #create arrays for dc voltage & current
dci_points = []
HOST = ''                                           #gets default hostname in setup
PORT = 50010                                        #arbitrary port
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)#create socket object
sock.bind((HOST, PORT))                             #bind socket object to port
print('socket bound')
sock.listen(1)                                      #listen for connections
print('listening')
clientSocket, address = sock.accept()               #form connection when esp joins
print("Connection established")
while True:                                         #while active
    for i in range(8):                              #sleep for 30 seconds
        sleep(1)
        print('sleep time: '+str(i))
    data = 'meas'                                   #create measurement query string
    clientSocket.send(data.encode())                #query measurement from pico
    recvData = clientSocket.recv(2048)              #anticipate 2kb response
    dataLen = len(recvData)                         #find length of response
    parsedData = dataParser(recvData.decode('utf-8'))   #decode and parse data
    for i in range(len(parsedData[0])):             #break parsed data into new arrays
        dcv_points.append(float(i))
        print(parsedData[0][i])
    for i in range(len(parsedData[1])):
        dci_points.append(float(i))
        print(parsedData[1][i])
    #plotter.plot(dcv_points,parsedData[0], label = 'dcv')
    #plotter.plot(dci_points,parsedData[1], label = 'dci')
    #plotter.legend()
    #plotter.show()
    dcv_points = []                                 #reset arrays after reading data
    dci_points = []
```

angle_finder.py for testing of IMU calibration

```
import time                         #enables sleep
import board                         #imports i2c code
import adafruit_icm20x                  #imports imu code
import math

i2c = board.I2C()                    #create i2c object
imu = adafruit_icm20x.ICM20948(i2c)     #create imu object

cal_values = open("imu_cal.txt","r")    #open file with calibrated values
#following pulls in front/back and side/side calibrated values and converts str to float
#assumes front/back is first, and side/side is second
#assumes lines are separated by \r
angle_frontBack_cal = float(cal_values.readline())
angle_sideSide_cal = float(cal_values.readline())

while True:                         #while true
    accel_data = imu.acceleration   #get data from accelerometer

    #Code below takes accelerometer data in m/s^2 and translates to an angle
    #calculates as mounted in custom enclosure
    #frontBack is y/z
    #sideSide is y/x

    angle_frontBack = ((180/math.pi)*(math.atan2(accel_data[1],accel_data[2])))+90
    angle_sideSide = ((180/math.pi)*(math.atan2(accel_data[1],accel_data[0])))+90

    #below code finds angle until rv is level
    #for frontBack, negative means rear end is lower than front end
    #positive means front end is lower than rear end
    #for sideSide, negative means right side from front is lower than left side
    #positive means left side from front is lower than right side

    angle_frontBack_toLevel = angle_frontBack_cal - angle_frontBack
    angle_sideSide_toLevel = angle_sideSide_cal - angle_sideSide

    #below code gives calculated angle, only relevant for testing
    print("front to back angle is ", angle_frontBack)
    print("side to side angle is ", angle_sideSide)
    print("front to back angle to level is ",angle_frontBack_toLevel)
```

```
    print("side to side angle to level is ",angle_sideSide_toLevel)
    time.sleep(1)               #sleep for a second while testing to avoid overrunning the
buffer
```

angle_cal.py for calibration testing

```
import time                         #not really necessary, but is here for testing purposes
import board                        #contains info for IMU object
import adafruit_icm20x              #has resources for i2c IMU
import math                         #used for atan2 and pi

i2c = board.I2C()                   #make i2c object
imu = adafruit_icm20x.ICM20948(i2c)         #make IMU object

angle_frontBack_data = []           #create array for front/back dataset
angle_sideSide_data = []            #create array for side/side dataset
angle_frontBack_sum = 0             #create variable for front/back sum for
average
angle_sideSide_sum = 0              #create variable for side/side sum for average
angle_frontBack_cal = 0             #create variable for calibrated front/back
average
angle_sideSide_cal = 0              #create variable for calibrated side/side average
cal_samples = 100                   #give number of calibration samples

for i in range(cal_samples):        #populate arrays for sample storage
    angle_frontBack_data.append(i)
    angle_sideSide_data.append(i)

for i in range(cal_samples):        #collect samples
    accel_data = imu.acceleration       #gets data from IMU, takes ~5ms per sample

    #uses atan2 to collect front/back and side/side angles
    #y direction points upward
    #z direction points forward
    #x direction points to side
    angle_frontBack_data[i] =
((180/math.pi)*(math.atan2(accel_data[1],accel_data[2])))+90
    angle_sideSide_data[i] =
((180/math.pi)*(math.atan2(accel_data[1],accel_data[0])))+90

for i in range(cal_samples):        #sums samples in arrays
```

```
        angle_frontBack_sum += angle_frontBack_data[i]
        angle_sideSide_sum += angle_sideSide_data[i]

#finds average value of collected readings
#divides sum of samples by number of samples
angle_frontBack_cal = angle_frontBack_sum / cal_samples
angle_sideSide_cal = angle_sideSide_sum / cal_samples

cal_values = open("imu_cal.txt", "w")          #open txt which holds calibrated values
cal_values.write(str(angle_frontBack_cal))      #write front/back average to file
cal_values.write("\r")                         #add newline, can use other delimiter depending
on operation
cal_values.write(str(angle_sideSide_cal))       #write side/side average to file
cal_values.close()                             #close file when done, good practice

print("Calibration Successful")                #indicate leveling is complete
```