Heliostat Control Tracking

Samuel Dixon
Jordan George
Erica Quist

Subsystem Report

REVISION – Draft A
December 2, 2022

Subsystem Report
FOR
Heliostat Control Tracking

Team <66>

Approved by:

Jordan George	12/2/2022
Project Leader	Date
Dr. Kalafatis	12/2/2022
Prof. Kalafatis	Date
Dalton Cyr	12/2/2022
T/A	Date

Change Record

Rev.	Date	Originator	Approvals	Description
A	12/2/2022	Samuel Dixon, Jordan George		Draft Release

Table of Contents

Table of Contents

No table of figures entries found. **5**

1. Introduction **6**
2. PCB Subsystem Report **7**
2.1. Subsystem Introduction 7
2.2. Subsystem Details 7
2.3. Subsystem Validation 9
2.4. Subsystem Conclusion 13

3. Data Visualization and User Interface Subsystem Report **14**
3.1. Subsystem Introduction 14
3.2. Subsystem Details 14
3.3. Phone Application 16
3.4. Web Application 17
3.5. Subsystem Validation 17
3.6. Diagnostic and Mitigation 20
3.7. Subsystem Conclusion 20

List of Tables

Table 1: Voltage Regulator with Varying Loads……...10
Table 2: Efficiency of the Voltage Regulator ……11
Table 3: Voltage at ESP32 with a varying input ……..11

List of Figures

Figure 1: PCB Schematic ……..……7
Figure 2: PCB layout ……..8
Figure 3: PCB with components soldered on ……9
Figure 4: Output Voltage vs Supply of Voltage Regulator ……9
Figure 5: Voltage Regulator of Varying Loads ...……..10
Figure 6: Oscilloscope Image of Input Voltage (12V) ...……..12
Figure 7: Oscilloscope Image of Output Voltage (3.3V)……..12
Figure 8: Front End of Website Application..…..……...14
Figure 9: Front End of Main Page of Phone Application……15
Figure 10: Front End of Manual Controls Page of Phone Application……..15
Figure 11: Front End of Data Analytics Page of Phone Application……16
Figure 12: Front End of Application Information Page of Phone Application ……...16
Figure 13: Code to programmatically check for Wifi connection……17
Figure 14: Code to programmatically write to database and type checking……..…..18
Figure 15: Code for callback method to get new updates and change table in web app…..19

1. Introduction

This project intends to develop the control methodology and design behind an energy application device known as a Heliostat. The device seeks to track the rotating motion of the sun throughout the day and rotate some mirrors to focus light onto a thermal plate for further energy conversion. The subsystems in this project entailed design and fabrication of a PCB, development of a user interface and data visualization, and tracking controls software for the microcontroller.

2. PCB Subsystem Report

2.1. Subsystem Introduction

The PCB subsystem intends to service the power to the microcontroller as well as provide a common interface for uploading code to the ESP32 Wroom through a micro USB connector as well as being able to access the GPIOs of the ESP32 Wroom for the various sensors of this project. The PCB will be powered by a 12V / 7A battery. There will be a voltage regulator part of this circuit to step down the 12V to 3.3V as the microcontroller and USB-UART bridge require around 3.3V to operate.

2.2. Subsystem Details

The PCB subsystem was created with the help of the PCB design software called Altium. Below is the schematic drawing of the PCB:

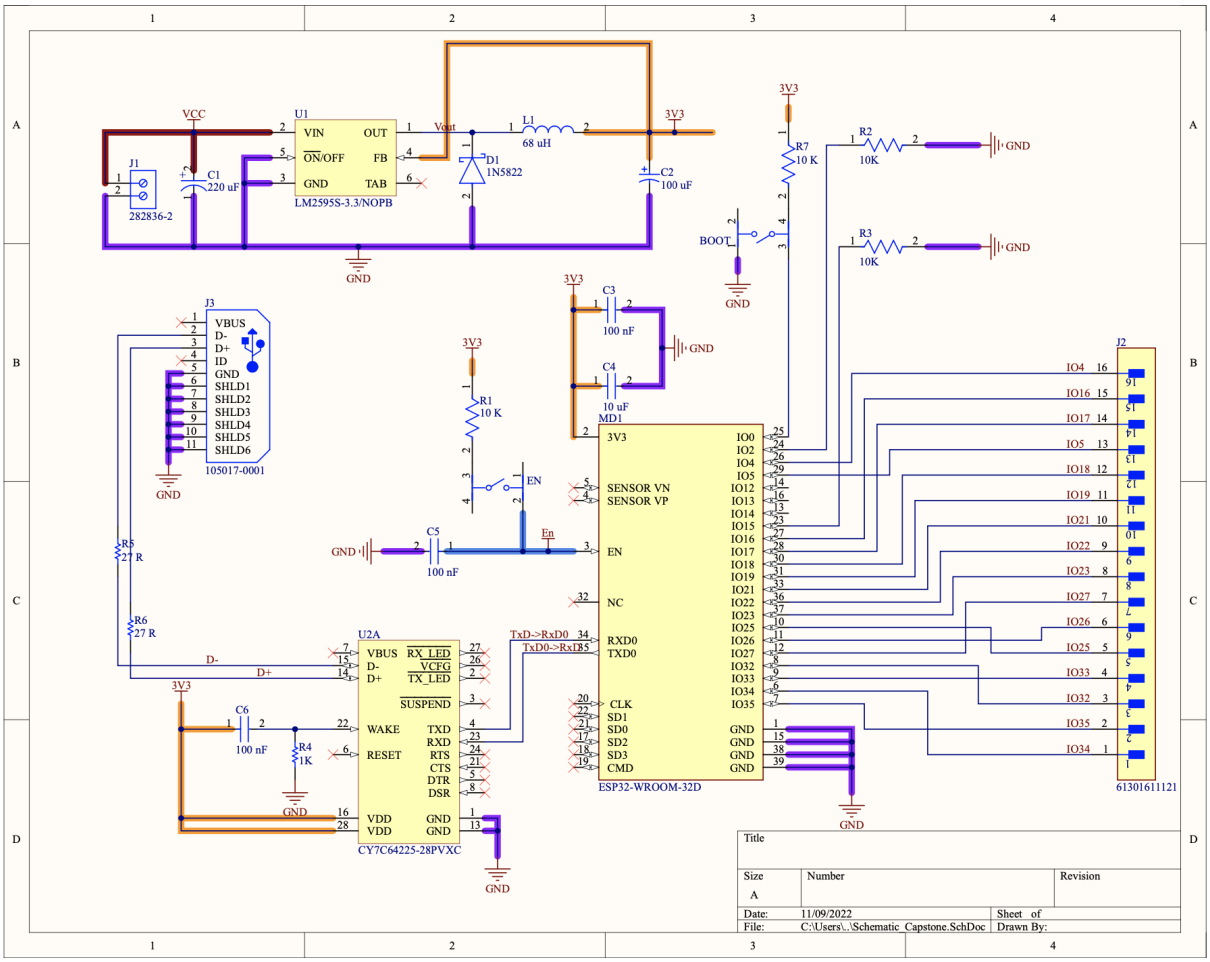

Figure 1: PCB Schematic

In the top left corner of the schematic is the voltage regulator that will step down the input voltage of 12V to 3.3V for the rest of this schematic. The input voltage of 12V is color coded by the dark red lines. The orange colored lines represent the 3.3V node while the purple colored lines represent the common ground node. The datasheet of the ESP32, voltage regulator, and the USB-UART bridge were read through to help best set up this schematic.

On the right side of the schematic are the 16 header pins that will be used for the multiple GPIOs of the ESP32. These will be hooked up to the different sensors in this project. In order to turn the ESP32 into bootloader mode, the BOOT button must be held down and shortly after, the EN button will be pressed. After, both buttons may be released. The next part of design process was the PCB layout pictured below:

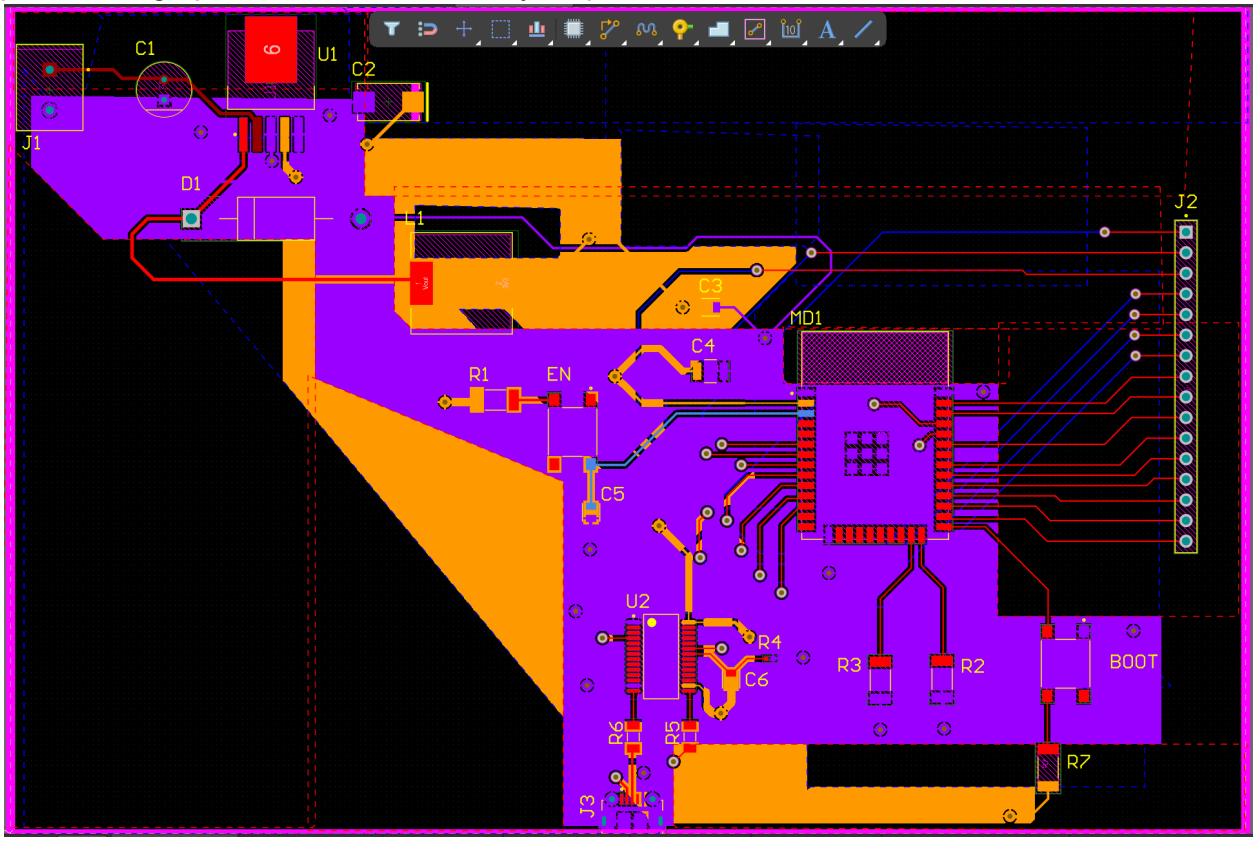

Figure 2: PCB layout

This layout starts with the screw terminals named J1 in the top left of this picture. This is where the input power will be plugged into. The board was able to be designed as a 2-layer board. There are two polygon pours to create a common node for needed voltages. The orange polygon pour is on the bottom layer and is the 3.3V net, while the purple polygon pour is on the top layer and is the common ground net. This Altium design was sent to JLCPCB for manufacturing. Once the board got here, all the components were soldered where they were designed to be, and it resulted into the following.

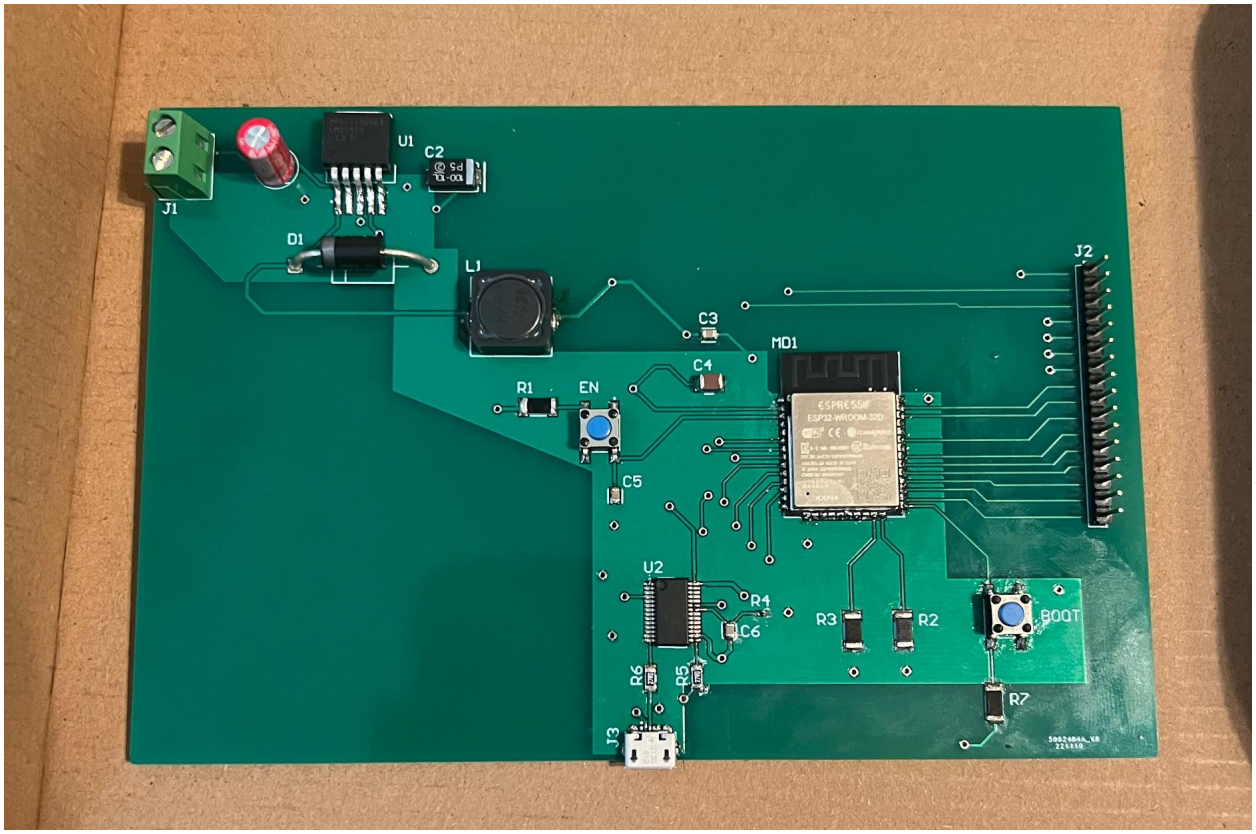

Figure 3: PCB with components soldered on

2.3. Subsystem Validation

The voltage regulator chip went under a lot of testing to see if it could hold a constant output of 3.3V. The first thing that was tested was varying the value of the input voltage.

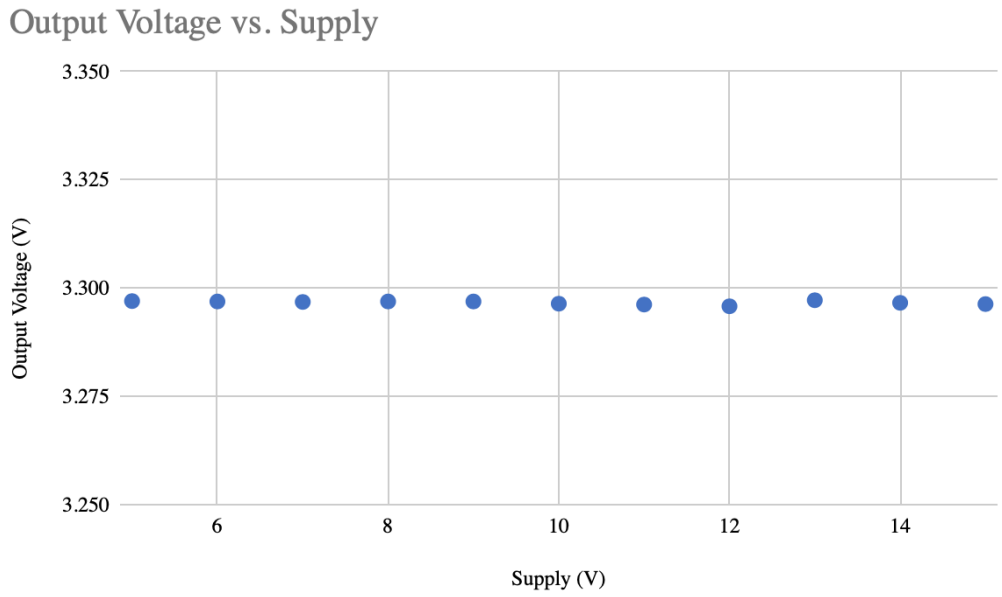

Figure 4: Output Voltage vs Supply of Voltage Regulator

As shown above, the output constantly holds slightly under 3.3. The nominal voltage of this experiment is 12V so this is working as desired, since the ESP32 and the UART bridge both operate at 3.3V. The next thing that was tested was by varying the load of the voltage regulator to see if the output still holds at 3.3V. The input voltage for these tests was held at a constant 12V.

Varying Load Tests			
Supply (V)	**Regulator Output (V)**	**Load being tested (Ohms)**	**Current At the Load (A)**
12	3.2958	Open circuit	0.006
12	3.263	15.83980583	0.206
12	3.227	7.665083135	0.421
12	3.193	5.295190713	0.603
12	3.155	3.933915212	0.802
12	3.122	3.144008056	0.993

Table 1: Voltage Regulator with varying loads

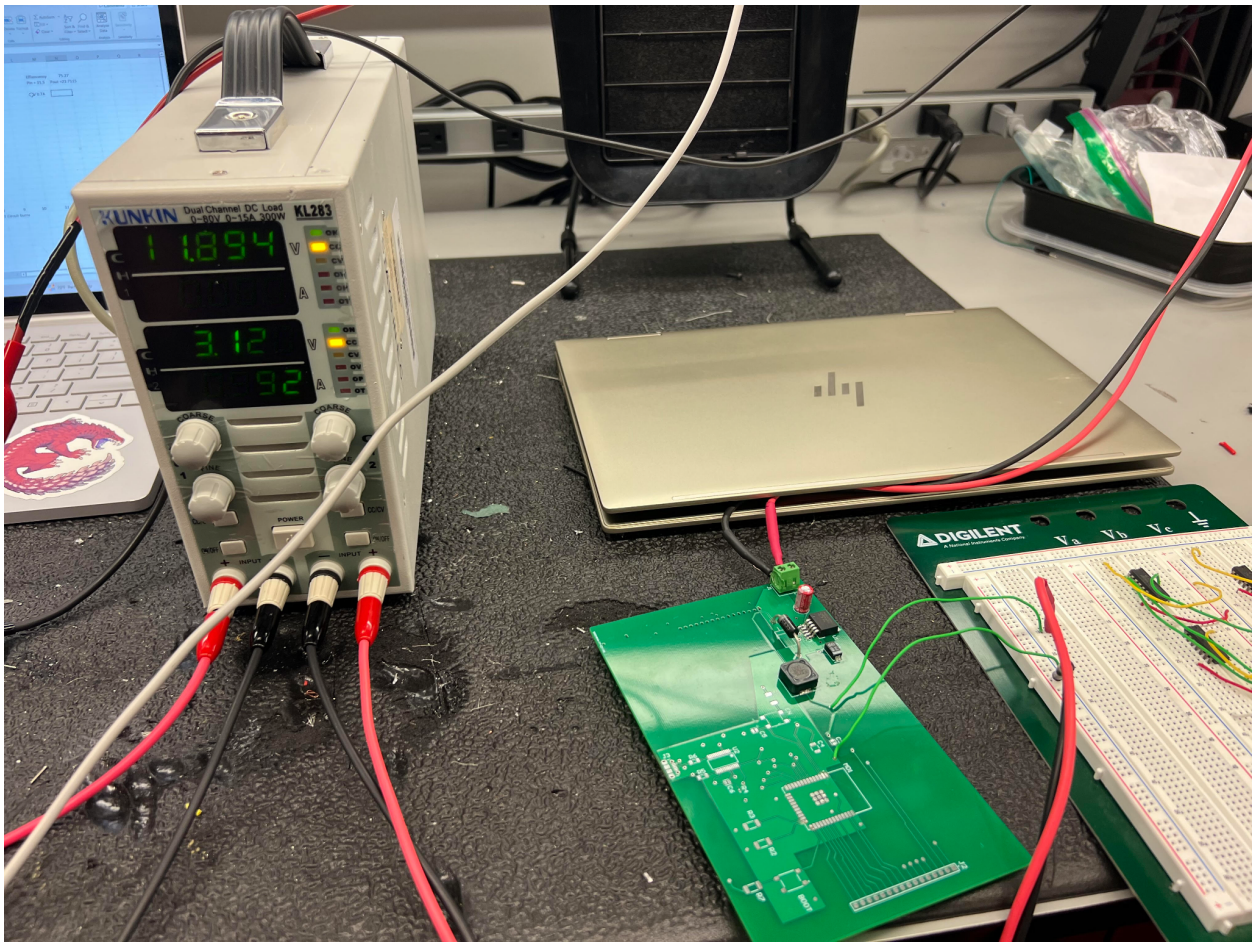

Figure 5: Voltage Regulator with varying loads

The max output current of the voltage regulator is 1 Amp so various data points were taken as the current drawn approached 1A. From Table 1, it can be seen that the voltage does

start to decrease slightly as the load decreases. The next set of data that was recorded was the efficiency of the voltage regulator. This was done by using one of the data points from Table 1, specifically the third row of the table. The input current was recorded by seeing how much current was being drawn from the power supply which was about 0.152 Amps.

Power In (W)	Power Out (W)	Efficiency
1.824	1.358567	0.7448283991

Table 2: Efficiency of the Voltage Regulator

The next set of testing was done on the circuit board that had all the components soldered on (Figure 3). First, it was tested to see if the input voltage would be stepped down to provide 3.3V to the ESP32. When the ESP32 was not in bootloader mode, almost no current was being drawn from the power supply. So the data in the following table was recorded while the ESP32 was on.

ESP32 when turned on		
Supply (V)	**ESP32 Voltage (V)**	**Current Drawn From Supply (A)**
5	3.2889	0.103
6	3.2893	0.086
7	3.29	0.08
8	3.3018	0.066
9	3.303	0.059
10	3.3022	0.057
11	3.2977	0.049
12	3.2978	0.045
13	3.2975	0.042
14	3.2968	0.04
15	3.2967	0.038

Table 3: Voltage at ESP32 with a varying input

This table shows that the circuit was working as expected. The output voltage would hold at a constant 3.3V which means the orange polygon pour from Figure 2 was at this voltage. This showed that the ESP32 and the USB-UART bridge were both being powered at 3.3V which is within their operating range. The last set of data recorded was taken by using the oscilloscope to measure the input and output voltage to check for any noise. Below are the pictures of what the oscilloscope showed for the input voltage of 12V and the output voltage of 3.3V.

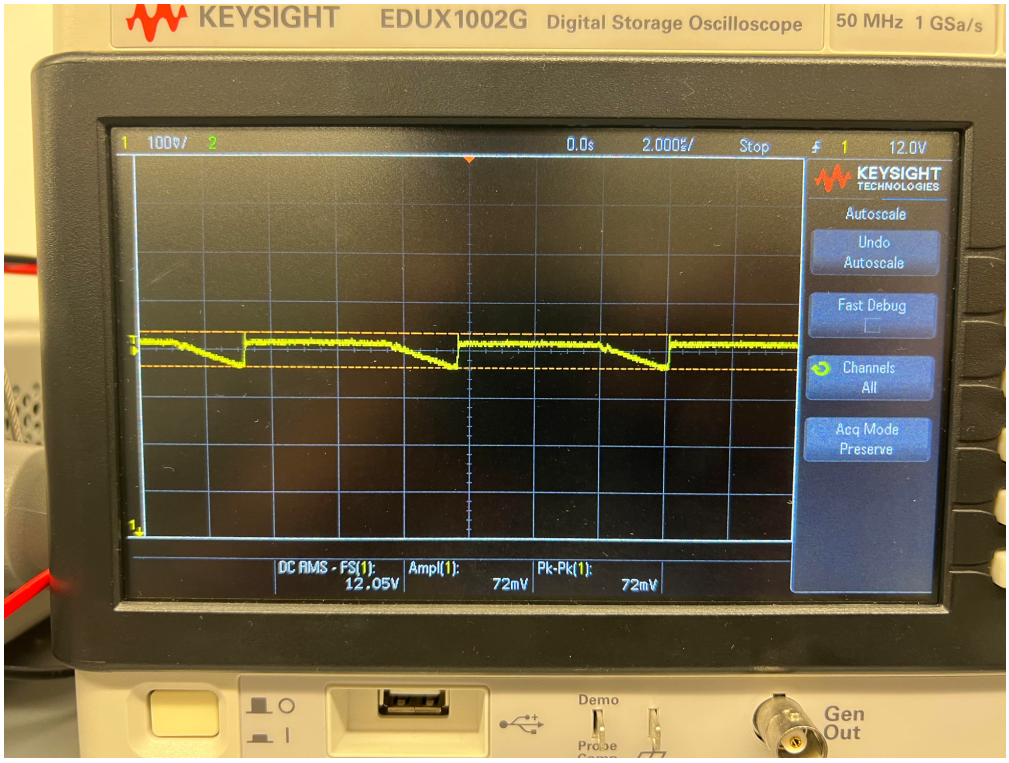

Figure 6: Oscilloscope Image of the Input Voltage (12V)

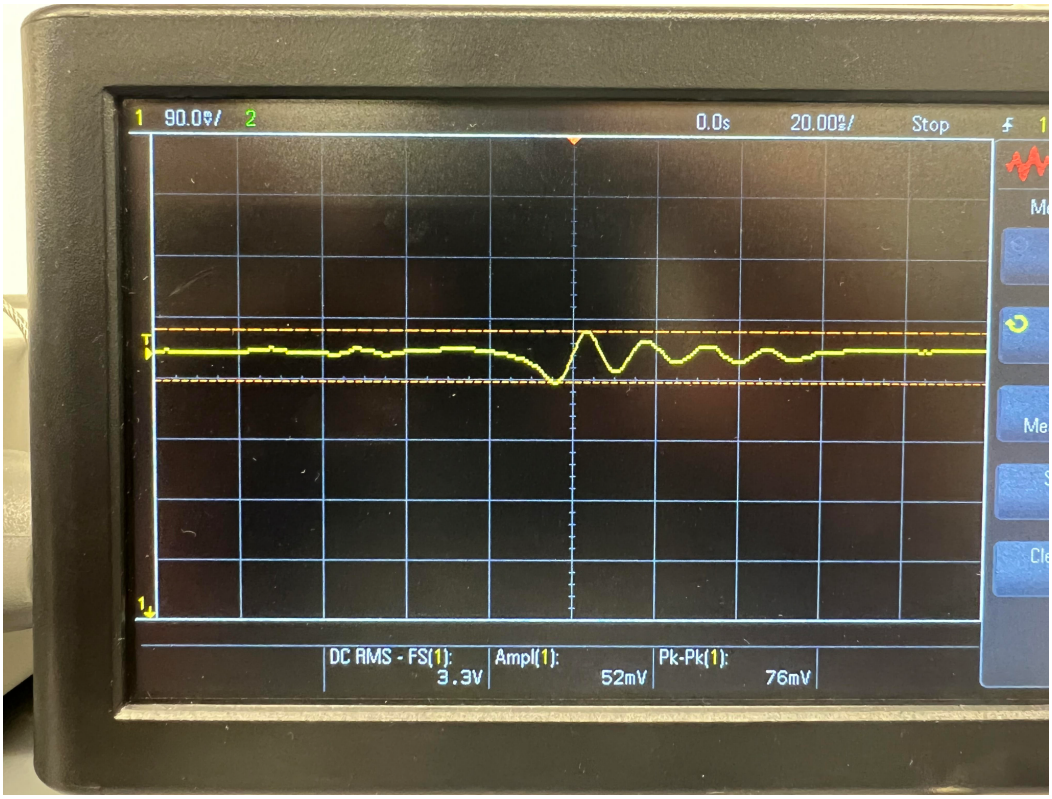

Figure 7: Oscilloscope Image of the Output Voltage (3.3V)

2.4. Subsystem Conclusion

From the validation tests shown in the section above, the PCB subsystem works as planned. This shows that the 12V battery that is being used for the whole project can be stepped down to power the ESP32 and USB-UART bridge. With the ESP32 and USB-UART bridge being operational, code can be written to these devices that will be able to control the motors of the project.

3. Data Visualization and User Interface

3.1. Introduction

The data visualization and user interface seeks to provide human machine interaction to the heliostat energy system. This interaction is done through use of a database, a web application, and a cellphone application. The database provides flags, which enable the user to interact with the motors in the heliostat on the cellphone application via wifi communication. The web application displays various sensor and timestamp readings for specific sensors from the database.

3.2. Subsystem details

This subsystem involved creating a cell phone and web application to provide a front-end visualization for the sensor readings stored in the online database of the system. This was done through using the google software suite. This was a convenient choice as it integrates their NoSQL non-relational, real-time database with android applications. Additionally, this software provides support for hosting websites on the cloud.

Heliostat Controls Web Data

	Sensor	Reading	Date
1	Photodiode1	25	Tuesday, November 29 2022 08:48:33
2	Magnetometer2	83	Tuesday, November 29 2022 08:48:33
3	Magnetometer1	15	Tuesday, November 29 2022 08:48:33

Figure 8 Website Table

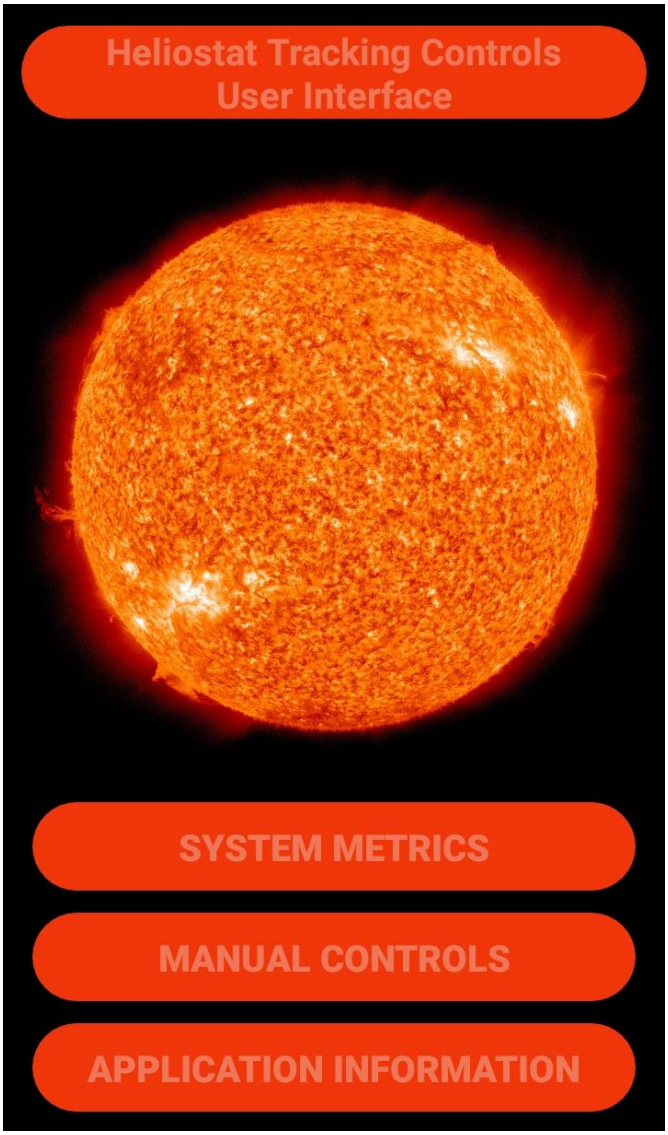

Figure 9: Main Activity Page

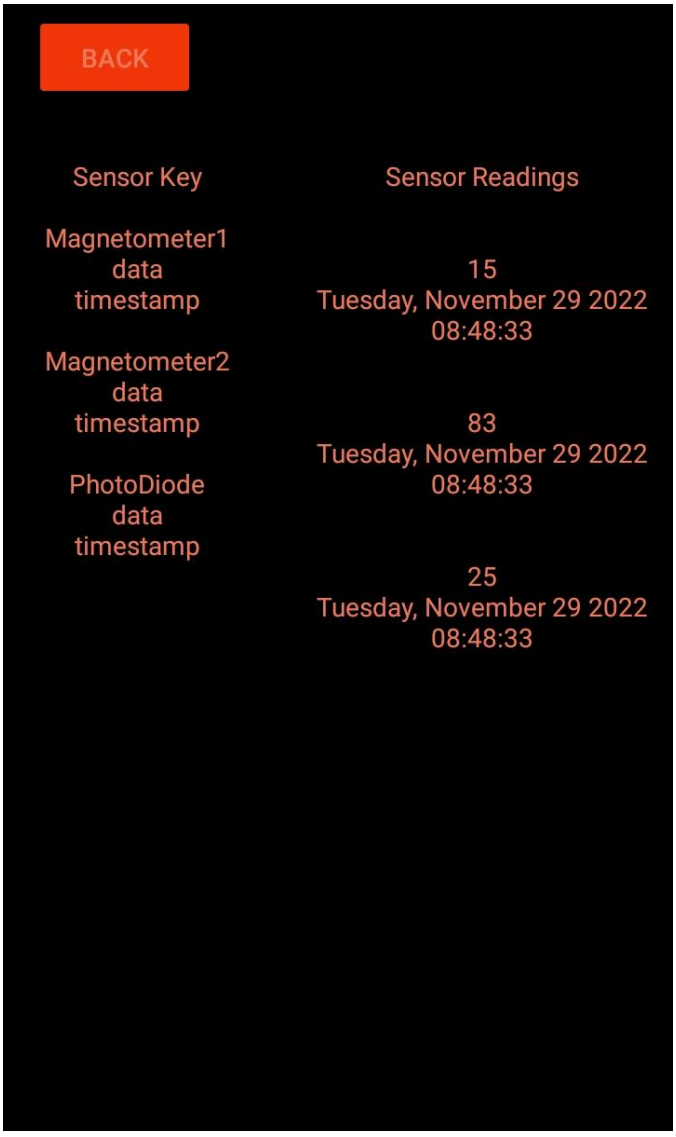

Figure 10: System Metrics

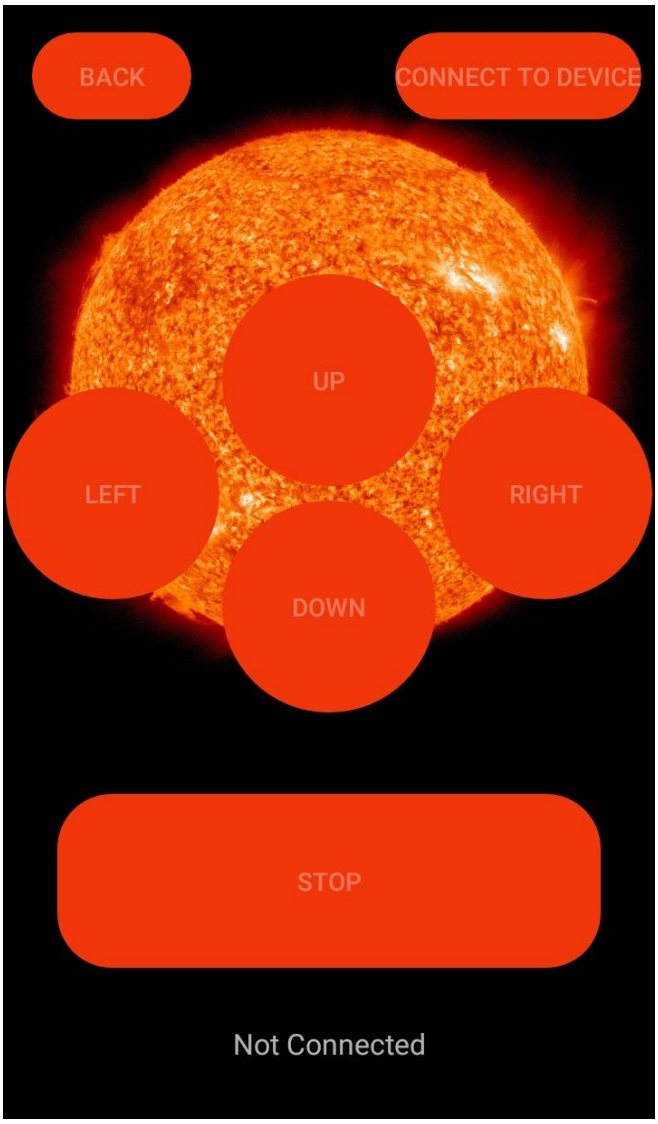

Figure 11 Manual Controls

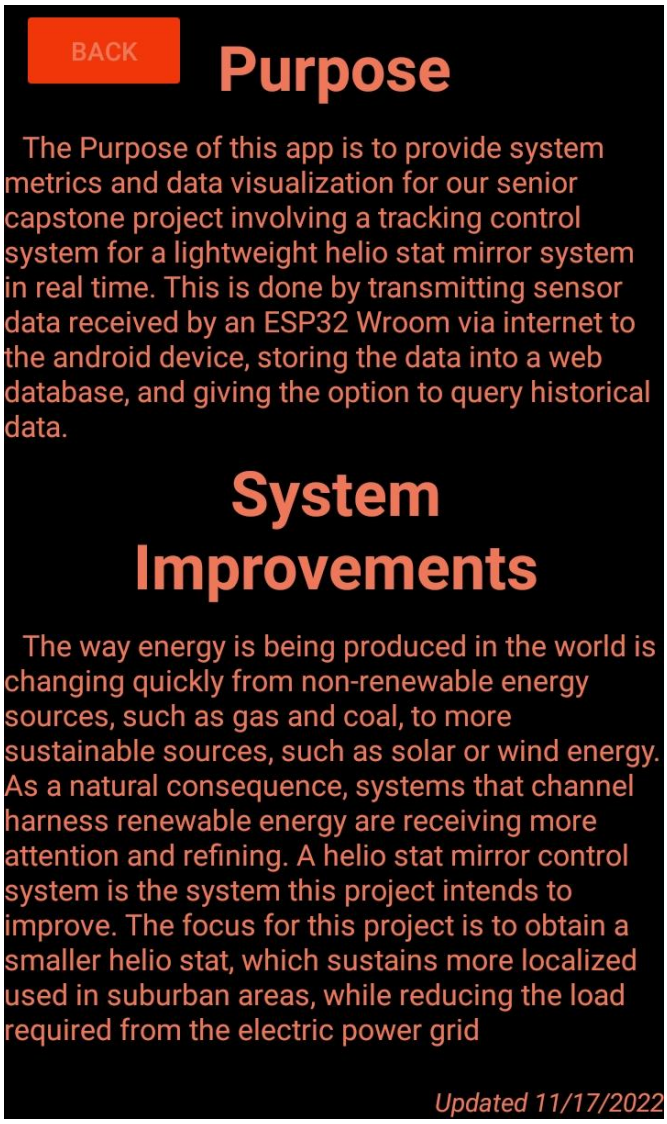

Figure 12: Application Information

3.3. Database

The firebase real-time database was used to store sensor data and service communication between the microcontroller and cellphone application. This database is a NoSQL non-relational database, which assumes a tree-like structure. Under the main parent node, there are two child nodes, Flags and Sensors. Flags represent the boolean flags, which can serve as service routine requests between the phone application and the esp32 microcontroller. Sensors contain all of the stored data sent from the esp32, which enables visualizing data analytics in both the web and phone application.

3.4. Phone Application

The phone application will provide user interface and data visualization for the system. The user interface is within the manual controls activity and provides users the opportunity to manual control the motors. There are two motors that have two directions of motion, so there are consequently four buttons to control the motors. Additionally, there is a connect to device button that acts as a handshake between the esp32 microcontroller and the phone application. These buttons all act through flags within the NoSQL database, and the esp32 is continuously querying these flags to change state. The data visualization is within the system metrics activity and provides users the opportunity to see the sensor readings in a text view display. This data comes from the Sensors node of the database as the most recent readings. Further room could be to have historical graphs within the application, rather than instantaneous readings.

3.5. Web Application

The Web Application is a simple website to also display the instantaneous data readings for the sensors of the system. This website is integrated with the database and the phone application within the firebase google software suite and is deployed on google's cloud hosted servers. The website involves html, javascript, and css, which use google's data visualization API to store the sensor data, which is a simple table seen in the figure below. This table also allows for column alphanumeric sorting, seen through the tabs at the top with the arrows.

3.6. Validation

Validation for these applications was performed by adding additional code to ensure the system functioned as intended. For example, in the phone application, in the system metrics and manual controls page, there must be an established wifi connection to interact with the database. This is seen in *Figure 13* below.

```
if (!connect) { // if no wifi, create alert dialog to inform user this function is necessary for application functionality
    builder = new AlertDialog.Builder( context: this);
    builder.setMessage("Connect to wifi or quit")
            .setCancelable(false)
            .setPositiveButton( text: "Connect to WIFI", (dialog, id) -> startActivity(new Intent(Settings.ACTION_WIFI_SETTINGS)))
            .setNegativeButton( text: "Quit", (dialog, id) -> openMain());
    AlertDialog alert = builder.create(); // creating alert builder
    alert.show(); // showing the content of the alert builder
}
```

```
private boolean haveNetworkConnection() {
    // initialize booleans assuming there is no wifi connection
    boolean haveConnectedWifi = false;
    boolean haveConnectedMobile = false;
    // Open connectivity manager and get the context for the system's wifi service
    ConnectivityManager cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
    // Create an array of network info to store the data from the connectivity manager getAllNetworkInfo method
    NetworkInfo[] netInfo = cm.getAllNetworkInfo();
    for (NetworkInfo ni : netInfo) {
        if (ni.getTypeName().equalsIgnoreCase( anotherString: "WIFI")) // Ignoring case (non case-sensitive) check for wifi
            if (ni.isConnected())
                haveConnectedWifi = true; // retract assumption, as network method informs otherwise
        if (ni.getTypeName().equalsIgnoreCase( anotherString: "MOBILE")) // Ignoring case (non case-sensitive) check for Mobile
            if (ni.isConnected())
                haveConnectedMobile = true; // retract assumption, as network method informs otherwise
    }

    if (!(haveConnectedWifi || haveConnectedMobile)) {
        Toast.makeText( context: ManualControls.this,  text: "Must connect to wifi to use this activity page", Toast.LENGTH_SHORT).show();
    }

    return haveConnectedWifi || haveConnectedMobile; // returning or for the two boolean states, as either implies a network connection
}
```

Figure 13: Programmatically checking for internet connection

Additionally, the full program flow and transition between activities and loading in the data from the database was confirmed through running the emulator in the android studio software suite and loading the application onto an android device. For the flag routine, this was done by first checking if the node existed in the database, and then ensuring that the node was of an integer type. This ensured that the correct data type was being written into the database and no type incompatibility problems ensued. In *Figure 14* this is seen below. The flag is first validated to be the correct type and then queried to get the integer at the database reference location. In *Figure 15*, a method was implemented to callback the table population function and create a new table that will allow for the updated sensor readings to be observed.

```
if (Firebase.RTDB.getInt(&fbdo, "Flags/Connect")) { // make sure that the connection flag has been set
  if (fbdo.intData() == 1) {  // if the reference is 1, then we can do movement
    if (Firebase.RTDB.getInt(&fbdo, "Flags/Right")) { // now accessing the firebase database object and setting the right flag
      if (fbdo.intData() == 1) { // if the value of the flg is 1

        Serial.println("right == 1 , clockwise"); // print serial data and note direction of motor

        Firebase.RTDB.setInt(&fbdo, "Flags/Right", 0); // reset the right flag to be 0

        myStepper.step(stepsPerRevolution); // use stepper module to actuate the movement of the motor

        Firebase.RTDB.setFloat(&fbdo, "Sensors/Magnetometer1/reading/data", random(1,100)); // testing a random datapoint to be set to the database
        update_time("Magnetometer1"); // call subroutine to update the time of the magnetometer1 sensor

        Firebase.RTDB.setFloat(&fbdo, "Sensors/Magnetometer2/reading/data", random(1,100)); // testing a random datapoint to be set to the database
        update_time("Magnetometer2"); // call subroutine to update the time of the magnetometer2 sensor

        Firebase.RTDB.setFloat(&fbdo, "Sensors/PhotoDiode/reading/data", random(1,100)); // testing a random datapoint to be set to the database
        update_time("PhotoDiode"); // call subroutine to update the time of the photodiode sensor

        Firebase.RTDB.setInt(&fbdo, "Flags/Right", 0);  // otherwise print 0 to indicate no movement

      } else {
        Serial.println("right == 0, no movement"); // otherwise we do not move and serial print to the screen
        delay(500); // delay 500 ms so that we can see what is happening logically
      }
```

Figure 14: Testing to write to firebase reference by first checking if getting an integer from flag returns true

```
onValue(sensors, (snapsot) => { // creating onValue to  reference the parent nodes of the sensors

    const sensor1 = ref(db, "/Sensors/Magnetometer1/reading/"); // getting reference to the reading of sensor1
    var d1, d2, d3, x, y, z; // instantiating variables to hold the values for the sensor data

    onValue(sensor1, (snapshot) => { // when any child of sensor1 changes, get new data
        x = snapshot.val()["timestamp"]; // child node key-value pair get time stamp
        d1 = snapshot.val()["data"]; // child node key-value pair get the data
    });


    const sensor2 = ref(db, "/Sensors/Magnetometer2/reading/"); // getting reference reading of sensor2

    onValue(sensor2, (snapshot) => { // when any child of sensor2 changes, get new data
        y = snapshot.val()["timestamp"]; // child node key-value pair get time stamp
        d2 = snapshot.val()["data"]; // child node key-value pair get data
    });

    const sensor3 = ref(db, "/Sensors/PhotoDiode/reading/"); // getting reference to the reading of sensor3

    onValue(sensor3, (snapshot) => { // when any child of sensor2 changes, get new data
        z = snapshot.val()["timestamp"]; // child node key-value pair get time stamp
        d3 = snapshot.val()["data"]; // child node key-value pair get data
    });

    google.charts.load('current', { 'packages': ['table'] }); // load the google chart api table
```

Figure 15: Implementing callback method to update the chart every time the sensor reading was changed

3.7. Diagnostic and Mitigation

For the sake of this project, the historical capture was determined not to be sufficient. There is no way currently to see a visualization of the previous sensor data. This will be implemented by adding a graph view page into the phone application. Additionally, the database organization has been determined to be insufficient for the sensor readings of the project. Two weeks will be taken to see if a revision of the current database will be sufficient for the purposes of this project, otherwise a new approach will need to be taken through utilizing a SQL database scheme.

3.8. Subsystem Conclusion

Through the validation and diagnostic and mitigation section of the report above, the subsystem has some functionality that is currently sufficient for the purposes of this project. However, there are still places where the subsystem needs to be improved for integration. This includes database organization and historical capture.