VALENCIACOLLEGE

Department of Electrical and Computer Engineering Technology (ECET)
Division of Engineering, Computer Programming, and Technology (ECPT)

EET 4950

Senior Design Proposal

The Smart Cooler

Submitted by

Clarence Scott and Reuben Taveras

Supervised by

Dr. Ali Notash

July 2, 2021

Abstract

Coolers are simply used as a portable way to keep things cool, whether it be drinks or food, among other things. Coolers are mainly designed to retain ice, but what if it could do more than that?

Given the state of IoT devices that are blooming in this generation, why not a cooler? By adding a power source to a cooler, it opens the device up to new, smart possibilities. The Smart Cooler features rechargeable batteries that can be charged from a 120V AC outlet, a 12V car port, or through a solar panel. Charging devices can easily be done with wireless or wired charging. A locking mechanism prevents the Smart Cooler from being opened by animals or unexpected visitors. The locking mechanism also can use RFID to activate using a keychain or card. An LED strip activates when it is dark or manually to provide light inside. GPS tracking shows the last known location of the Smart Cooler, allowing someone to find their campsite. Bluetooth speakers are built into the cooler to provide entertainment for any scenario. The cooler has two separate compartments for item storage, each with its own temperature sensor. Each compartment can be set to have a notification appear when the ice has melted in the cooler. A touchscreen GUI allows the user to monitor temperature, battery life, and settings. The features can be controlled, and temperature monitored through a Bluetooth app, which also functions with the GPS system. The cooler has savable profiles so allow for easy setting configurations. Finally, for added convenience, the cooler will have a flashlight holder and bottle opener.

Having a cooler with these kinds of capabilities has many advantages and provides an all-in-one solution for any outdoor event. This cooler is designed to be taken to the beach, to the woods, and just any backyard party.

Acknowledgements

We would like to thank Professor Notash for his assistance and guidance throughout the proposal process. The weekly meeting following suggestions and improvements to the project really helped to set apart our idea from any other out there.

Table of Contents

Abstract .... *ii*

Acknowledgements .... *iii*

List of Figures .... *vi*

List of Tables .... *ix*

Chapter 1 Introduction **1**
1.1 Introduction 2
1.1.1 Motivation 2
1.1.2 Objective and Features 2
1.1.3 Similar and Existing Products 4
1.1.4 Smart Cooler Survey 6
1.2 Proposed Systems 7
1.2.1 Block Diagram 8
1.2.2 Design Engineering Requirements 9
1.2.3 Design Engineering Specifications 10
1.3 Organization of Report 18

Chapter 2 Background Research **19**
2.1 Introduction 20
2.2 Software 20
2.2.1 Python 20
2.2.2 React Native 20
2.2.3 Bluetooth Technology 21
2.3 Hardware 22
2.3.1 Cooler 22
2.3.2 The Lid 34
2.4 Power Budget 36

Chapter 3 Contribution **38**
3.1 Introduction 39
3.2 Design Integration and Implementation 39
3.2.1 Power System 39
3.2.2 Device Charging 43
3.2.3 Controller 45
3.2.4 LED Lights 46
3.2.5 GPS 48
3.2.6 Locking Mechanism 49
3.2.7 Bluetooth Speaker Entertainment System 51
3.2.8 Touchscreen 52
3.2.9 Mobile App 57
3.2.10 Software 62

3.3 Design Troubleshooting and Results 65
3.3.1 Troubleshooting 65
3.3.2 Results 67
3.4 Discussion of Success and Failure Rates 82

Chapter 4 Non-Technical Issues 85
4.1 Budget 86
4.2 Timeline 88
4.3 Environmental Aspects 89
4.4 Health and Safety 89
4.5 Ethical Aspects 90
4.6 Social Aspects 90
4.7 Sustainability 90

Chapter 5 Conclusion 91
5.1 Summary and Conclusion 92
5.2 Suggestions for Future Work 92

References 93

Appendix A: Equations 96

Appendix B: Smart Cooler Survey 97

Appendix C: Smart Cooler Survey Results 99

Appendix D: Flowchart 102

Appendix E: Raspberry Pi Software Code 104

Appendix F: React Native Software Code 147

Appendix G: Lessons Learned 174

Biography 178

List of Figures

Figure 1.1 The INFINITE Smart Cooler 5
Figure 1.2 The Coolest Cooler 5
Figure 1.3 Smart Cooler Survey Question 1 Results 6
Figure 1.4 Smart Cooler Survey Question 3 Results 7
Figure 1.5 Block Diagram 8
Figure 2.1 Smart Cooler Dimensions 23
Figure 2.2 Smart Cooler Top View 23
Figure 2.3 Smart Cooler Back View with Rendered Environment 24
Figure 2.4 Coleman 100qt Cooler 24
Figure 2.5 Raspberry Pi 4 25
Figure 2.6 Pyle Low Profile Marine Speakers 26
Figure 2.7 DS18B20 Waterproof Temperature Sensor 26
Figure 2.8 CdS Photoresistor 27
Figure 2.9 Micro SG90 Servo 28
Figure 2.10 Duty Cycle of PWM Signal for SG90 Servo 28
Figure 2.11 NOCO Genius 10 29
Figure 2.12 Eco-Worthy 12V 10W Solar Panel 29
Figure 2.13 Miady LFP16AH 12V 16Ah LiFePO4 Rechargeable Battery 30
Figure 2.14 BN-880 GPS Module 30
Figure 2.15 Car Port USB Charger 31
Figure 2.16 Alitove WS2812B RGB LED Strip 31
Figure 2.17 RC522 RFID Kit with Blank Card and Key 32
Figure 2.18 12V to 5V 8A USB Buck Converter 33
Figure 2.19 DROK 12V to 5V 5A Buck Converter 33
Figure 2.20 FQP30B06 N-Channel Power MOSFET 34
Figure 2.21 Qi Wireless Charging Transmitter 35
Figure 2.22 SunFounder 7” Touch Screen 35
Figure 2.23 Reed Switch 36
Figure 3.1 12V 16Ah Batteries 40
Figure 3.2 Terminal Blocks 40
Figure 3.3 Lithium Battery Charging Device 41

Figure 3.4 Solar Panel 42
Figure 3.5 Solar Panel Charging Module 42
Figure 3.6 Running Wires Through Interior Panel 43
Figure 3.7 Wires Run Through Wall 44
Figure 3.8 Wired Charging Port 44
Figure 3.9 Wireless Charging Pads in Lid 45
Figure 3.10 Blue LEDs, Rainbow LEDs, Red LEDs, White LEDs 47
Figure 3.11 Photosensor Mounted on Exterior of Cooler 48
Figure 3.12 GPS Mounting Bracket 48
Figure 3.13 Google Maps Using GPS Coordinates 49
Figure 3.14 Locking Mechanism without Covers 50
Figure 3.15 Holes for Speakers 51
Figure 3.16 Bluetooth Speakers Turned On 51
Figure 3.17 Speaker Amp Board 52
Figure 3.18 Touchscreen GUI Main Screen 53
Figure 3.19 Touchscreen GUI Settings Screen One 54
Figure 3.20 Touchscreen GUI Settings Screen Two 55
Figure 3.21 Touchscreen GUI Ice Monitor Compartment One Activated 56
Figure 3.22 Touchscreen GUI Ice Replacement Notification 56
Figure 3.23 Mobile App Scanning Screen 57
Figure 3.24 Mobile App Connect Screen 58
Figure 3.25 Mobile App Main Screen 59
Figure 3.26 Mobile App GPS Location Screen 60
Figure 3.27 Mobile App Settings Screen 61
Figure 3.28 Mobile App Ice Notification Compartment One 62
Figure 3.29 3D Printed Sliding Lock 65
Figure 3.30 Smart Cooler Front View 67
Figure 3.31 Smart Cooler Back View 67
Figure 3.32 Smart Cooler Side View 68
Figure 3.33 Smart Cooler Top View 68
Figure 3.34 Smart Cooler Interior View 69
Figure 3.35 Both Compartments Filled with Food and Ice 71
Figure 3.36 Cooler Outside with No Ice in Compartments Temperatures 71
Figure 3.37 Cooler Outside with Ice in One Compartment Only Temperatures 72

Figure 3.38 Cooler Outside with Comp. One Ice and Food Only Temperatures..........72
Figure 3.39 Cooler Outside Both Compartments with Ice Temperatures.....................73
Figure 3.40 Cooler Out. Comp. One w/ Ice and Food and Comp. Two w/ Ice Temp ..73
Figure 3.41 Cooler Outside Both Compartments Ice and Food Temperatures 74
Figure 3.42 Cooler Inside with No Ice in Compartments Temperatures 74
Figure 3.43 Cooler Inside with Ice in One Compartment Only Temperatures 75
Figure 3.44 Cooler Inside with Comp. One Ice and Food Only Temperatures 75
Figure 3.45 Cooler Inside Both Compartments with Ice Temperatures........................76
Figure 3.46 Cooler In. Comp. One w/ Ice and Food and Comp. Two w/ Ice Temp. 76
Figure 3.47 Cooler Inside Both Compartments Ice and Food Temperatures................77
Figure 3.48 Temp. Measurements in Comp. One and Comp. Two 77
Figure 3.49 Solar Panel Ratings 78
Figure 3.50 Solar Panel Open Circuit Voltage During Partly Cloudy Day 79
Figure 3.51 Solar Panel Short Circuit Current During Partly Cloudy Day...................79
Figure 3.52 Solar Panel Open Circuit Power During Partly Cloudy Day 80
Figure 3.53 iPhone Charging Percentage Over Time..81
Figure 3.54 Android Charging Percentage Over Time 81
Figure 4.1 Graphical Representation of Timeline 78

List of Tables

Table 1.1 Design Engineering Requirements 10
Table 1.2 Design Engineering Specifications 11
Table 2.1 Power Budget 36
Table 3.1 Raspberry Pi Pinout 46
Table 3.2 System Operation Current Measurements 70
Table 3.3 Speaker Decibel Rating 82
Table 3.4 Success Criteria 82
Table 3.5 Final Testing Results 83
Table 4.1 Actual Project Budget 86
Table 4.2 Senior Design Proposal Timeline 88
Table 4.3 Senior Design Project Timeline 88

Chapter 1

Introduction

Summary

The Smart Cooler is designed to be a multipurpose all-in-one cooler used for parties, get-togethers, or any social gathering. The cooler features speakers, wireless and USB charging, an RFID locking mechanism, GPS tracking, rechargeable batteries, multiple external charging options, touchscreen interface, mobile app support using Bluetooth, interior LED lights, cup and flashlight holder, and bottle opener.

1.1 Introduction
1.2 Proposed Systems
1.3 Organization of Report

1.1 Introduction

Coolers are an essential part of any sort of outdoor gathering. Whether it be camping, enjoying the beach, or tailgating, the idea is to have nice cold drinks on a warm summer day. Coolers do a good job at maintaining temperatures and allowing ice to last longer, and ideally that's all they have been designed for. However, they can be designed to do more. The Smart Cooler was designed to have a plethora of functionality, tailored for any sort of outing. The goal of The Smart Cooler was to make an all-in-one cooler, bringing together accessories that are usually bought for gatherings but in one easy to use product.

1.1.1 Motivation

When planning for any social gathering, food and entertainment are the number one accommodation to allow people to enjoy the gathering. Usually they include music, so Bluetooth speakers were added. For drinks, to keep them cool, it would be ideal to notify the user when the ice has depleted. At the beach, for instance, there are no power sources, so having a power bank adds a level of convenience. Most of the cooler utility is to keep food and drinks cold, but once you add power to the cooler the applications can now expand beyond that.

1.1.2 Objective and Features

Once power is provided to the cooler, the possibilities open for more functionality. At the beach, ideally drinks would be ice cold and opened with a bottle opener while music plays through waterproof Bluetooth speakers. The risk of bears looking for food can be prevented with a locking mechanism for the cooler. If the cooler has GPS functionality and the group becomes lost while exploring, the cooler location can be used to find the campsite. At night, an interior LED could make it easier to find things inside. App connectivity suddenly turns an ordinary cooler into a Smart cooler, where the temperature can be monitored. All of these features will be powered by rechargeable batteries that can be recharged through an outlet, a 12V car port, or solar panel.

Entertainment

Bluetooth speakers attached to the cooler provide ample entertainment for any outing scenario. The speakers will be waterproof as they will be used for outdoor speakers.

Power Source

Since typically coolers are used in outings, using the cooler power source as a charging bank for portable devices adds another utility to those who will spend extended periods of time from a power source. Wireless charging as well as wired charging will be provided.

RFID Locking Mechanism

Animals have a keen sense of smell and tend to rummage around looking for food. Providing the cooler with a RFID locking mechanism prevents food and drinks from being scavenged out of the cooler. It also prevents unwanted guests from opening the cooler as well.

GPS Tracking

Since coolers tend to be left at the campsite when camping, providing the user a means to locate the campsite via the cooler provides survival functionality. GPS does not require data or internet, so it may be used in the woods.

Rechargeable Batteries

The batteries powering the cooler require an ample amount of amperage. Therefore, the cooler will be given rechargeable batteries as a renewable resource of energy.

Multiple External Charging Options

The rechargeable batteries will be able to be charged through a 120V AC outlet, a 12V car port, or a 12V solar panel.

Touchscreen Interface

A graphical user interface will be designed to be used on a touchscreen interface for intuitive use. The temperature will be displayed for each compartment, along with accessing settings.

App Support

Temperature monitoring and setting will also be accessible via a mobile app designed for both Android and iPhone support. The mobile app will interact via Bluetooth to communicate with the microcontroller.

Night Light

Given there are no lights in locations like the woods, the cooler will have its own LED light built into the inside. A sensor will sense when it is dark outside and activate the light when the cooler is opened.

Structural Utilities

The cooler will have cup holders, as well as a bottle opener. A flashlight holder will also be built into the cooler.

1.1.3 Similar and Existing Products

Similar products have been produced with tremendous interest and success. Looking at crowdfunding sites like Kickstarter and Indiegogo, two Smart coolers that stand out are the INFINITE Smart cooler and The Coolest Cooler.

INFINITE

The INFINITE Smart cooler can be found on Indiegogo. The campaign has been closed, raising $136,106 out of its flexible goal of $25,000 [1]. The cooler features a 550W blender, wireless charging, LED light, HD built-in camera, Hi-Fi dual-powerful Bluetooth speakers, digital screen and easy-touch buttons. It has foldable side handles, large sturdy wheels, side storage, multifunction utensils and cutting board, and 61 quarts capacity. The cooler claims 7 to 12 days of ice retention [1].

This cooler highlights the importance of functionality. Coolers are used for party outings and can function as an all-in-one cooler.

Fig. 1.1: The INFINITE Smart Cooler [1]

The Coolest

The Coolest Cooler is considered one of the most successful Kickstarter of 2014, and also considered one of the biggest Kickstarter disasters. The campaign raised $13,285,266 from 62,642 backers [2]. Ultimately, the founder was unable to provide 20,000 backers with coolers, blaming tariffs imposed on products imported to the US from China [3]. However, this Kickstarter highlights the demand for a multifunctional Smart cooler with diverse applications.

This cooler featured an 18V battery powered rechargeable blender, removable waterproof Bluetooth speakers, USB charger, LED lid light, gear tie-down, cooler dividers and cutting board, extra wide easy rolling tires, integrated storage for plates and knife, and bottle opener.

Fig. 1.2: The Coolest Cooler [2]

1.1.4 Smart Cooler Survey

A survey was conducted as a sample size to represent the general population. The survey is shown in Appendix B with each question asked. The survey resulted in 47 responses, and the responses can be found in Appendix C. The first question of the survey asked which feature the participants would be most excited about.

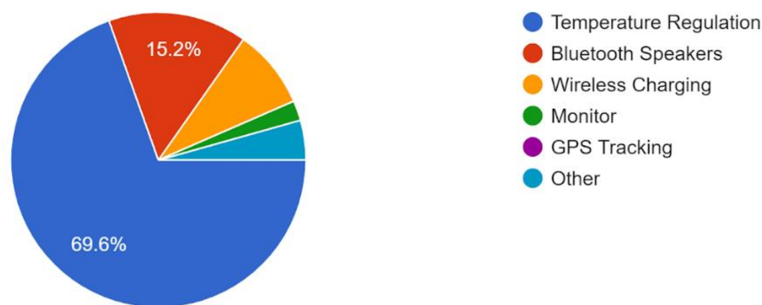


Fig. 1.3: Smart Cooler Survey Question 1 Results

The results show that out of the six choices, the most dominant desired feature is temperature regulation, with 69.9% of the votes. Second and third were Bluetooth speakers and wireless charging, respectively. The INFINITE Smart cooler and The Coolest Cooler both do not have any sort of refrigeration system for temperature regulation besides insulation. Originally, our design could implement thermoelectric refrigeration units, however, given advice from the panel, it would be impractical to design a product that uses such components as they are heavy on power usage and will not give us the desired results. Therefore, the other features were focused on instead.

The second part of the survey asked what features would be desired for a Smart cooler that wasn't mentioned. The results can be summarized as follows:

- Wheels that work on multiple surfaces
- Temperature monitoring through an app or digital screen
- Removable freezer blocks that can create dividers, shelves, and integrate into the lid
- Notification when the ice melts
- Maintain different temperatures for different compartments

- GPS tracking and a way to contact emergency services
- Screen to play games
- LED light inside

Though there were already plans to implement some of these features such as temperature monitoring through an app or different temperatures for different compartments, the other suggestions were nice additions that were considered for the project. Several of these proposed ideas were not implemented in the two most successful crowdfunded campaigns mentioned earlier. Therefore, valuable data was obtained from the survey.

Finally, to have a general idea of what the cooler would most often be used for, the survey asks where the participant would use the cooler most often.

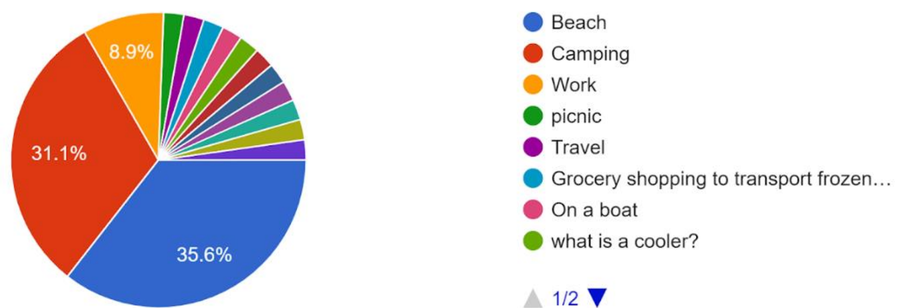


Fig. 1.4: Smart Cooler Survey Question 3 Results

Most of the results were split between camping and the beach as the places the cooler would be used most often. With these results, the focus of the project was to obtain features that would be most beneficial to people mainly to those kinds of outings. The question was also open for suggestions, which show other uses such as for picnicking, travel, grocery shopping, etc.

1.2 Proposed Systems

Planning of the Smart Cooler was done by starting with a block diagram. Design engineering specifications and design engineering requirements show the justifications for the parts selected for the modules. Each part is described in detail to highlight their contribution to the overall functionality of the project.

1.2.1 Block Diagram

The block diagram shows each module that the cooler will have. There are seven modules in total, as shown in Figure 1.5, represented by the colored blocks. Figure 1.5 is a block diagram which details how the different modules will work together to create the entire system.

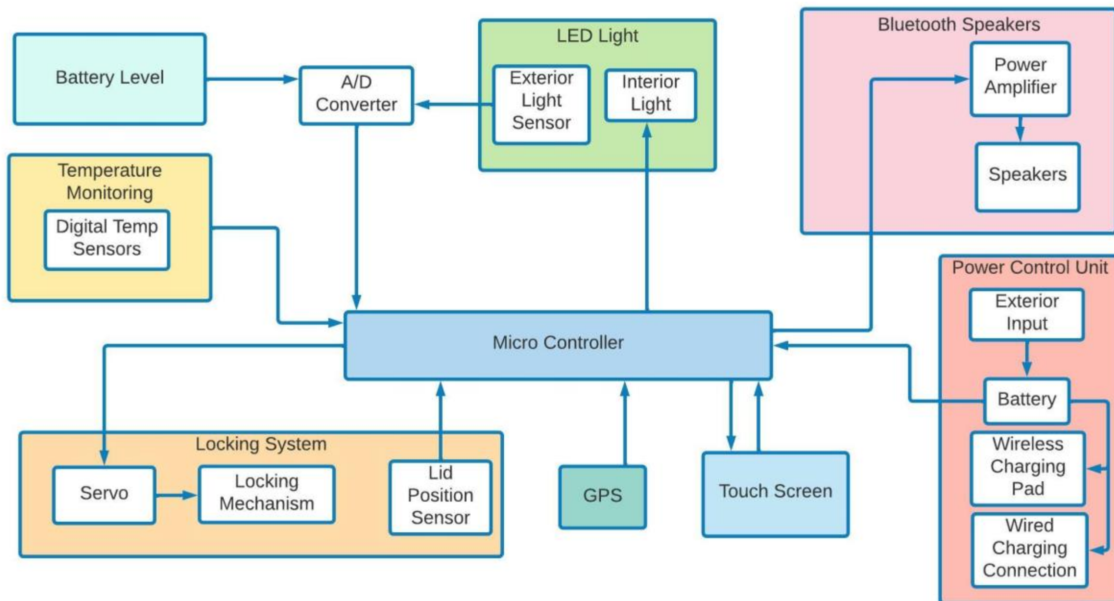


Fig. 1.5: Block Diagram

All the modules will be communicating with the microcontroller, as that will be the "brain" of the system. The microcontroller will be a Raspberry Pi, that contains 40 pins which can be used as GPIO pins and other communication protocols. At least 20 pins have been mapped out for all modules. The Raspberry Pi will either be acting as a switching device for larger power sources or will be used with the communication protocols to send and receive information.

The temperature monitoring module contains digital temperature sensors that are waterproof and embedded into the frame of the cooler to provide accurate temperature readings. The microcontroller will then determine whether to notify the user based on the temperature readings.

The LED light module also has a sensor that detects the amount of light outside. The analog to digital converter is used so that the Raspberry Pi can read the sensor values. If the value is determined to indicate a dark environment, the LED light strip will be

turned on by the microcontroller. The battery level indicator also makes use of the analog to digital converter.

The GPS module will be sending its location to the microcontroller. That information will be relayed to the Bluetooth app so that the location can be recorded on the map. With the last known location recorded, if someone were to wander off, they would be able to use the GPS position along with a compass to find the location.

The touch screen will not only display a GUI to the user but will also provide feedback to the microcontroller. Through the touch screen, the temperature will be able to be monitored, the LED light can be turned on or off manually, the locking mechanism can be activated, and the speakers can be turned on or off. The user can configure things such as the light color, temperature units, auto features, setting and saving profiles, and setting ice notifications. All of this is configurable through a GUI menu.

The Bluetooth speakers will have a separate Bluetooth module as the Raspberry Pi will not be able to use audio since some of the features required will need to have audio disabled. When the user connects to the Raspberry Pi through Bluetooth, someone will be able to controller the cooler, while that same person or someone else can connect their phone to the speakers and play music. A power amplifier is required for the speakers as they are 100W speakers.

The locking mechanism will require a servo to rotate the mechanism into an open and close position. The locking mechanism will use a reed switch to check if the lid has been closed all the way and will lock if properly closed.

All of this will be powered by two 12V batteries. The power control unit provides charging to the batteries by external sources such as an AC outlet, car port, or solar panel. The batteries will provide power to all the modules, as well as power for any device connected to the USB charging ports or wireless charger.

To see the program logic of how the microcontroller will be communicating with all these modules, the flowchart can be found in Appendix D.

1.2.2 Design Engineering Requirements

The Design of the Smart cooler **shall** include the following required features:

Table 1.1
Design Engineering Requirements

High-Level Requirements	• Temperature monitoring for range minimum of 0 degrees Celsius (Temperature of ice) accurate to within 1 degree Celsius. • Minimum active operation of all systems 12 hours. • Multiple exterior charging from 120V AC outlet, 12V car port, or 12V solar panel. • Mobile app functionality including features such as temperature monitoring, notification system if temperature gets out of range, activating locking mechanism and interior LED, GPS location tracking, and settings. • 7" touch screen GUI allowing for temperature monitoring, alert system, activating locking mechanism and interior LED, and settings. • Maximum empty weight of 40 lbs. • Contains a volume of about 1.5 cubic feet for storage.
Mid-Level Requirements	• Wireless/Wired charging options at 5V, 1 to 3A. • GPS Tracking with accuracy within 3 meters. • RFID keyless entry locking mechanism. • Bluetooth Speakers allowing for minimum 87 dB. • Solar Panel integration may provide 12V 1.5A charging power to battery.
Low-Level Requirements	• Cup holders standard minimum 2" diameter. * • Wheels capable of handling rugged terrain. * • Storage space. * • Interior LED activated when lighting conditions outside are dark. • Flashlight holder. • Bottle opener. *Denotes features that will be covered in the Shell of the cooler

1.2.3 Design Engineering Specifications

Table 1.2 outlines the engineering specifications of the system's modules and individual components.

Table 1.2
Design Engineering Specifications

COOLER				
Module	**Specific Components**	**Engineering Specification**	**Justification and Verification**	**Responsibility**
Shell	Coleman 100qt	Should have cup holders, wheels, and be large enough to support multiple compartments, at least 1.5 cubic feet of space. Exterior walls should be at least 2' thick. Maximum allowable empty weight of 40 lbs.	**Justification:** Anyone should be able to move or load the Smart Cooler. **Verification:** Smart Cooler should be lightweight, have large carrying capacity, and wheels to help with transportation and loading.	Clarence
Control	Microcontroller: Raspberry Pi 4	Controller will provide SPI (Serial Peripheral Interface) used for communicating with other boards or modules. At least 20 GPIO pins needed, with alt functionality for SPI and serial communication. Bluetooth will connect with app at a range of	**Justification:** The Pi 4 can handle the input and output traffic and communicate with the other modules. While connecting to the HDMI touch screen or mobile app. **Verification:** The microcontroller will be able to handle the six primary inputs, and four	Team

		15m, and microcontroller will operate between 3.5-5.5V.	primary outputs, and communicate with the other modules. The microcontroller has 27 GPIO pins total with up to 6 alternate functions. Testing will ensure the microcontroller is communicating with the application.	
Mobile Application	Apple or Android Smartphone	Bluetooth will communicate with microcontroller at a range of 15m. Application will monitor temperature of each compartment, ability to activate locking and interior LED, ability to change settings (GPS enable, locking mechanism enable, LED light enable, speaker enable), and able to save profiles.	**Justification:** Providing Android and iPhone support allows for the majority of users to use the app. The app communicating with the microcontroller allows for remote monitoring and setting of the cooler. **Verification:** Testing will ensure the microcontroller is communicating with the mobile application using an Apple or Android Smartphone.	Reuben

Entertainment	Pyle Marine Speakers	Speakers should be no deeper than 2”. Speakers should operate between 50-200W for a dB rating of 87 dB.	**Justification:** Narrow footprint so that they can be placed in the walls of the Smart Cooler. Speakers must be heard when near the cooler. **Verification:** Speakers should be able to be heard at least 5 meters away.	Clarence
	Amp: TPA3116 DAMGOO	Provide at least 200W for Speaker operation. Use class D amplifier for maximum efficiency.	**Justification:** Exceed minimum Wattage needs for both Speakers. **Verification:** Speakers should properly operate with the wattage provided from the AMP.	Clarence
Sensors	Temperature: DS18B20 Waterproof Temperature Sensors	Monitor Temperature in compartments to an accuracy of ± 1.0°C in real time, minimum range of at least 0°C	**Justification:** The temperature sensor will tell when the compartment is too warm and when ice needs to be replaced. **Verification:** Monitor the temperature with a separate thermometer in the compartments.	Reuben

Interior LED	Exterior Light: Photo-sensitive Sensor Alitove Led Strip Lights	Exterior light sensor will change output voltage in low light conditions in real time. LED light strip will provide illumination when light sensor detects it is dark outside.	**Justification:** Interior lights are needed for operation after dark. **Verification:** Will test and confirm that output voltage changes when ambient light is low. Light strip will activate when dark outside or manually.	Reuben
Power Input Control Module	Voltage Regulator 12V input power socket	Shall provide the cooler multiple charging options to accept 120V AC and 12VDC.	**Justification:** Allow for battery charging from Solar Panel, outlet plug, or automobile barrel jack. **Verification:** A digital multimeter will be used to confirm that the module is supplying the correct voltages when both inputs are used to pass.	Team
Battery	Battery: LiFePO4	Provide 12V to the Smart Cooler modules. Battery shall maintain active operation for at least 12 hours.	**Justification:** Needed to power systems when outside power is not available. **Verification:** A digital multimeter will be used to	Team

		To be charged by charging module.	confirm that the battery is supplying the correct voltages to pass.	
	Solar Panel: Eco-Worthy 12V 10W	12-20V output voltage, at least 1.5A current output, at least 14.4W power output, at least 9.6Ah output	**Justification:** The solar panel will provide power to the battery during daylight hours and assist with charging for nighttime operation. **Verification:** A digital multimeter will be used to confirm that the module is supplying the correct voltages when both inputs are used to pass.	Clarence
Power Output Control Module	Voltage Regulator	Convert stored energy in the battery into voltages required for operation, 5V and 12V.	**Justification:** The Voltage Regulator will ensure that the voltages supplied by the batteries are the correct voltages to ensure the modules of the Smart Cooler operate properly without damaging components.	Team

<table>
<tr><td></td><td></td><td></td><td>Verification:
A digital multimeter will be used to confirm that the voltage regulator is allowing the correct voltages to pass.</td><td></td></tr>
<tr><td></td><td>Exterior USB Charger: Damavo YM1218 USB C and USB A Charger socket</td><td>Should operate using either 5V or 12V input voltage.

Will output 5V 2.1A for USB A &
5V 3A for USB C</td><td>Justification:
The charger will allow wired charging of devices.

Verification:
Plug will be tested with several USB devices</td><td>Clarence</td></tr>
<tr><td colspan="5">LID</td></tr>
<tr><td>Module</td><td>Specific Component</td><td>Engineering Spec</td><td>Justification</td><td>Responsibility</td></tr>
<tr><td>GPS Module</td><td>BN-880 GPS Module</td><td>Module will accurately track location to within 3 meters.</td><td>Justification:
Allow the user to mark the Smart Cooler and possible camp site.

Verification:
Google Maps will be used to confirm the accuracy of the GPS location.</td><td>Team</td></tr>
</table>

Charging Module	Qi Wireless Charging Transmitter	Output at 5W at a minimum of 100 KHz Provide 5V and 1A of charging power to devices.	**Justification:** Allows the user to charge other mobile devices in a timely manner. **Verification:** Capable of charging modern Smart devices especially cell phones.	Clarence
Lock	Locking Mechanism Sensor: Reed Switch Servo: SG90	Close and lock the Smart Cooler using the APP touch screen or RFID Keyless entry. Provide 180 degree rotation to position lock in place.	**Justification:** Secure the lid so that it cannot be opened on accident, unauthorized people, or animals. **Verification:** The lock should engage and disengage when signaled by the Touch screen, Mobile app, and RFID keyless entry.	Team
Touch Screen	Sunfounder 7 inch	Shall support touchscreen functions and at least Wide SVGA resolution.	**Justification:** Allows the user to interact with the Micro controller and control the operation of the cooler. **Verification:** Touching screen interacts with GUI and sets or configures	Clarence

			features on cooler.	

1.3 Organization of Report

The report is split into five chapters. Chapter 1 involves the general overview of the project. It mentions the problem trying to be solved, the motivation, and the proposed solution, along with similar solutions. The system requirements and specifications engineered for this project are also in this chapter. Chapter 2 is where the background research will be evaluated. Any concepts or theory learned will be placed in this chapter, as well as what hardware and software is needed, and the power budget. Chapter 3 is the contribution that went towards the design of the project, in other words what went into the implementation of the project, any troubleshooting that occurred, and the results with success and failure rates. Chapter 4 involves the non-technical aspects, such as the timeline followed for the project, the revised budget, environmental affects, health and safety, ethical aspects, social impact, and sustainability. Chapter 5 gives the conclusion for the report. References are listed after Chapter 5, followed by Appendices.

Chapter 2

Background Research

Summary

Any background research conducted for the project is given in-depth. The various software and hardware used, with reasoning for why each component is used and a detailed explanation of its purpose are organized in this section. The power budget is also given.

2.1 Introduction
2.2 Software
2.3 Hardware
2.4 Power Budget

2.1 Introduction

After going through the design engineering requirements and specifications, the desired parts detailed in this chapter were chosen to fit into the design of this project. The Raspberry Pi and mobile phone app both used different programming languages, so separate languages needed to be learned. Since the app would be using Bluetooth, that also needed to be researched, as that is a field in itself. The cooler features many components, and those components are highlighted in this chapter. The power budget was analyzed to help choose the right batteries for the project.

2.2 Software

In order to make use of the GPIOs and communication protocols of the Raspberry Pi, Python needed to be learned. The mobile app needed to be created for both Android and iPhone, so React Native was the technology used to develop for both platforms. The phones needed to communicate wirelessly with the Raspberry Pi, so Bluetooth needed to be learned.

2.2.1 Python

Python is the most common programming language used for the Raspberry Pi. Many of the libraries found to interface with the hardware devices were written in Python. A Python library needed to be found for the MCP3008 ADC, the WS2812B addressable LEDs, the BN-880 GPS module, the SG90 servo, and the RC522 RFID module. Since Python is the most common language, it makes it easier to find resources for connecting with other peripherals. Not only is Python used to communicate with the different peripherals, but also used to control the GPIO pins to set them high or low, and program the look of the GUI interface. Even Bluetooth communication was done using Python, even though Bluetooth can be done using the terminal.

2.2.2 React Native

Given that the project requires a mobile app that can function on both the Android and iPhone platforms, instead of learning the native code for both mobile phones and writing and maintaining the code for two separate platforms, a single platform was used instead. React Native implements JavaScript code that gets built into both Android and iPhone

code. The advantage of using React Native is that the code can be written, modified and maintained in the same language, but can be deployed in both major environments. To program for the Android, the code is compiled from JavaScript into native Android code and can be deployed to a test Android phone or Android simulator. In order to develop for the iPhone, a MacBook must be used. The development environment XCode allows for React Native to be compiled and deployed to a test iPhone or an iPhone simulator.

2.2.3 Bluetooth Technology

When devices need to send and receive information without the use of Wi-Fi, Bluetooth tends to be the standard set for close distance communication between devices. Bluetooth tends to be used to stream audio or send and receive messages between devices. There are two types of Bluetooth: Bluetooth Classic and Bluetooth Low Energy.

Bluetooth Classic vs Bluetooth Low Energy (BLE)

Both Bluetooth Classic and Bluetooth Low Energy use the 2.4 GHz frequency band to send and receive information. However, there are differences between the two, thus each having different uses.

Bluetooth classic provides two-way communication with an application throughput of 2.1 Mbps [4]. It’s highly effective at short distances. Bluetooth Classic is mainly used to stream audio and is commonly used for wireless speakers, headphones, and in-car entertainment systems [5]. It has a latency of 100 ms and up to 30 mA of power consumption [4].

Bluetooth Low Energy has an application throughput of 0.3 Mbps, with a limit of 20 bytes packages allowed to be sent [4]. Bluetooth Low Energy is used for its communication capabilities but is now also used for indoor positioning [5]. It has low power consumption (up to 15 mA, but can be 100x lower), with a range of 100 m and a latency of 3 ms [4].

Bluetooth Properties

Bluetooth’s main advantage is being able to communicate with devices without a router or access point. To follow the Bluetooth standard, the Bluetooth device must have a profile defined so it can communicate with other devices. A GATT or general access profile (GAP) defines the records of Services, Characteristics, and Descriptors. Services

are the data structure that contain Characteristics. Characteristics contain information such as its type, value, properties, and permissions. The permissions allowable are read, write, and notify. Read and write are self-explanatory, but notify means that when the Characteristic changes, the device connected to it will be notified of the change and read the value. Descriptors are just a general description of the Characteristic and is optional. Each attribute must have a UUID, either a 16-bit value or a custom 128-bit UUID [6].

Device discovery involves the defining general access profile (GAP), where the device that needs to be discovered is “advertising” its GAP, and the device scanning for the advertising device is searching and connecting to it. Advertising involves sending packets continually, so that the device scanning can detect it. The advertising device is the Bluetooth peripheral, and the scanning device is the Bluetooth central [6].

2.3 Hardware

Given the specifications of the project, these parts were found to accomplish the tasks for each module. A lot of fabrication will be involved, so general hardware and any 3D printed components are not included.

2.3.1 Cooler

The cooler will have a structure that allows for ice retention and for all the electronics to go into. The walls will be built with plastic with some kind of insulation to allow better temperature regulation. The dimensions of the cooler will be 17” x 36.4” x 17.9”, shown in Figure 2.1.

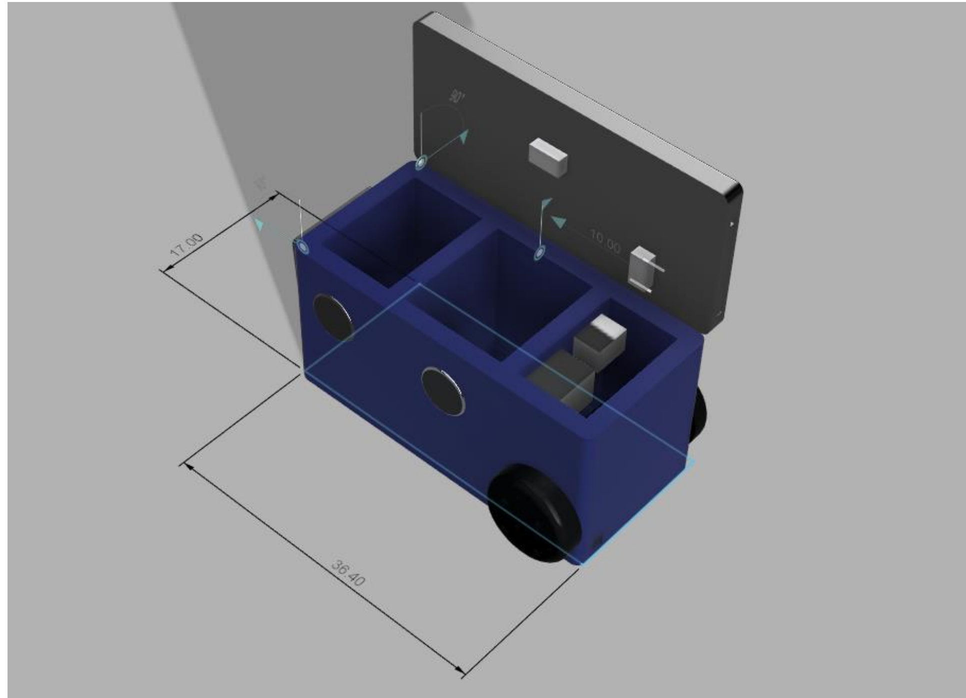


Fig. 2.1: Smart Cooler Dimensions

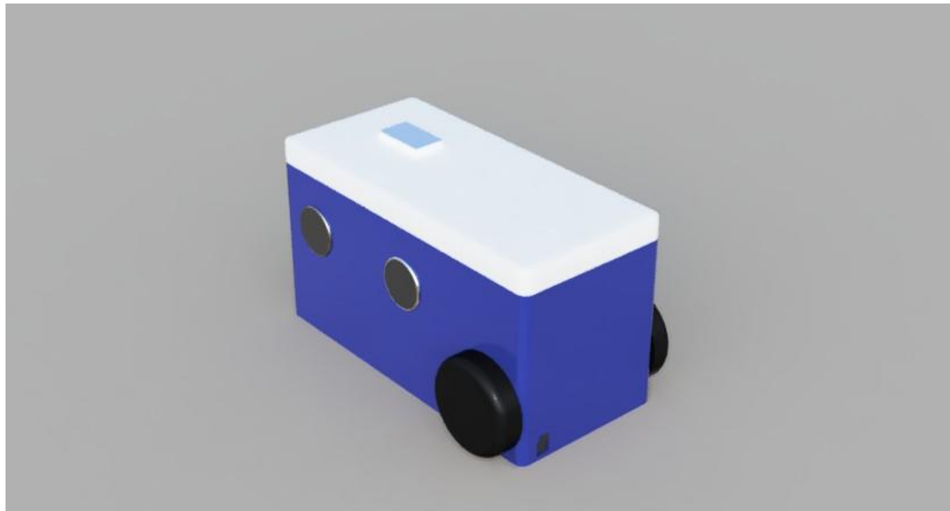


Fig. 2.2: Smart Cooler Top View

Fig. 2.3: Smart Cooler Back View with Rendered Environment

The Shell

The shell is the platform that everything will be constructed around and on. Since the cooler would require rugged wheels and a handle, the Coleman 100qt XTREME 5 Day Cooler was selected as the shell. The cooler lid contains cup holders, while a flashlight holder and bottle opener were added. It has the required wheels and is advertised as retaining ice for 5 days.

Fig. 2.4: Coleman 100qt Cooler [7]

The Microcontroller

The microcontroller is the brain for this system. It must provide serial communication, GPIOs, and SPI (Serial Peripheral Interface) used for communicating with other boards or peripherals. Bluetooth connectivity is needed to support the use of a mobile application.

Fig. 2.5: Raspberry Pi 4 [8]

Speakers

Given that the speakers were built into the wall of the cooler, the speakers were narrow. Also given that there will undoubtedly be dealing with water and moisture around the cooler and compartments, the speakers needed to be waterproof. The selected speakers were the Pyle outdoor speakers.

Fig. 2.6: Pyle Low Profile Marine Speakers [9]

Temperature Sensors

The temperature sensor first and foremost must be able to reliably track the temperature in the compartments. Sensors like these are instrumental in telling the microcontroller the status of the compartments and environment so that it can correctly control the activation of the refrigeration unit. Given that the readings are required to be fast and in real-time with an accuracy of ± 1.0°C, the DS1820B waterproof temperature sensors were selected. The sensors have a rated accuracy of ± 0.5°C with a range of - 10°C -± 85°C. The module uses three pins: pin 1 is VCC (+3.3V), pin 2 is Data, and pin 3 is Ground.

Fig. 2.7: DS18B20 Waterproof Temperature Sensor [10]

The temperature sensor reads data in Celsius. To convert to Fahrenheit, the following equation is used.

Celsius to Fahrenheit Equation $$°F = ℃ \cdot \frac{9}{5} + 32 \quad (2.1)$$

Light Sensor

The Smart Cooler uses a photosensitive sensor to track the level of light in the outside environment because this will be what triggers the activation of the internal LEDS. Using the CdS photoresistor with an analog to digital converter gives the desired light readings.

The photoresistor is essentially a variable resistor that changes with light levels. When it is dark, the resistance is high, and when it is light, the resistance is low. Since the measured voltage will be in relation to the light levels, a voltage divider is used. The voltage across the resistor will change depending on the light levels, between 0 to 3.3V as that is what the Raspberry Pi outputs. The voltage level measured is converted to a digital value using the MCP3008 ADC.

Fig. 2.8: CdS Photoresistor [11]

Servo

The locking mechanism uses a hobby servo to rotate the horn and extend the dowel. The servo chosen is the SG90, with a torque of 2.5 kg/cm, as not a lot of torque is needed for the project. The application will only lock and unlock the locking mechanism, turning 90 degrees and staying in position.

Fig. 2.9: Micro SG90 Servo [12]

The servo pins are red (VCC +5V), orange (PWM), and brown (Ground). The SG90 PWM pin uses pulse width modulation to determine the position of the servo arm. The frequency of the PWM signal must be 50 Hz or 20 ms per cycle. The duty cycle, or on time, can vary between 1 to 2 ms, controlling the position of the arm from 0 to 180 degrees [13].

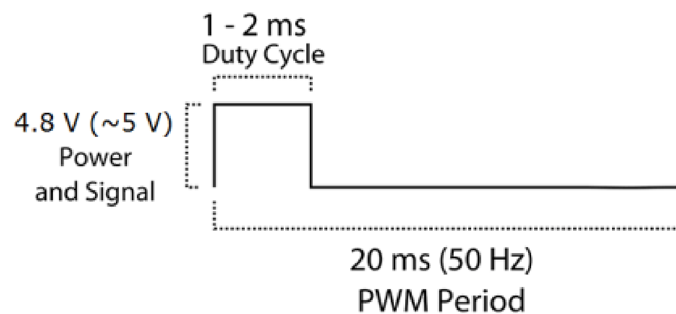


Fig. 2.10: Duty Cycle of PWM Signal for SG90 Servo [13]

Battery Charger

Given that the system runs on two 12V batteries wired in parallel, the correct battery charger needed to be found for the specific batteries selected. Looking at the datasheet for the Miady batteries that were selected, it was specified a charging current of 10 amps. Therefore, the NOCO Genius 10 battery charger was found. It provides charging for lithium batteries, as lithium batteries need a certain algorithm to charge fully. Lithium chargers are designed to use constant current constant voltage charging (CCCV) to fully charge a battery, where initially it uses a constant current to charge it up to 70%, then

switches to constant voltage, where the current decreases as it fully charges [14]. The NOCO Genius 10 provides 10 amps of charging current for 12V lithium batteries.

Fig. 2.11: NOCO Genius 10 [15]

Solar Panel

To improve the time that the active systems can be on we are using a solar panel to trickle charge the battery for the Smart Cooler. Because the batteries that will be in parallel have very specific charging requirements, we need a solar panel with circuitry that is compatible. The Eco-Worthy 12V 10W solar panel is small enough to use the space on the back of the shell, applies a 12V input, and is waterproof.

Fig. 2.12: Eco-Worthy 12V 10W Solar Panel [16]

Batteries

The Smart Cooler is powered by two rechargeable lithium batteries that are set up in parallel. This will functionally create one battery with twice the capacity. The LiFePO4 battery was chosen because it is a rechargeable battery that supports having its terminals

connected in parallel. Connecting two batteries with identical specifications allows double the Ah rating of the battery, while keeping it rated for the same voltage. The batteries are rated for a maximum discharge current of 42A, and a charging current of 10A.

Fig. 2.13: Miady LFP16AH 12V 16Ah LiFePO4 Rechargeable Battery [17]

GPS

The GPS module communicates with the microcontroller to record the last known location of the cooler. The GPS module chosen was the BN-880 GPS module, which also has a built-in compass. The pins for the BN-880 are as follows: pin 1 is SDA, pin 2 is GND, pin 3 is TX, pin 4 is RX, pin 5 is VCC (+3.6 ~ 5 V), and pin 6 is SCL. It is not required for GPS to have internet or data, as the GPS modules uses multiple satellite data to triangulate its position.

Fig. 2.14: BN-880 GPS Module [18]

USB Charger

One of the requirements of the Smart Cooler was to incorporate a wired charging option for Smart mobile devices. The plug supports both popular USB charging options USB A and USB C.

Fig. 2.15: Car Port USB Charger [19]

LED Light Strip

The LED strip is an RGB addressable light strip. The LEDs are able to change colors and can be programmed by the Raspberry Pi. The light strip wraps around the interior of the cooler and can be activated in auto mode when the light sensor detects it is dark outside, or manually set on and off. The LEDs used on the Alitove LED strip are the WS2812B LEDs, which are compatible with the Raspberry Pi, and can be programmed to change the colors of the LEDs.

Fig. 2.16: Alitove WS2812B RGB LED Strip [20]

The LED strip wires include VCC (+5V), data signal, and ground. Depending on the colors of the LEDs, the current draw can change. Each LED is rated for a max current consumption of 60 mA. Given the LED strip has 30 LEDs per meter, and a total of 2.25 meters is required to wrap around the cooler, the total amount of LEDs and max current consumption can be calculated.

$$Total\ LEDS = 30\frac{LED}{m} \cdot 2.25m = 68\ LEDs$$

$$Max\ Power\ Consumption = 68\ LEDs\ \cdot 60mA = 4.08A$$

RFID

This specific RFID module was chosen as it is compatible with the Raspberry Pi. The module comes with a blank card and key. Ideally, the key would be used to lock and unlock the cooler, as people must always have their car keys, but not their wallet. The RC522 module uses SPI to communicate with the Raspberry Pi. The pins include pin 1 is VCC (+3.3V), pin 2 is reset (RST), pin 3 is GND, pin 4 is interrupt (IRQ), pin 5 is MISO for SPI, pin 6 is MOSI for SPI, pin 7 is serial clock SCK for SPI, pin 8 is serial input (SS) for SPI. The module uses 13.56 MHz frequency, as that is the frequency allowed to operate without a license [21].

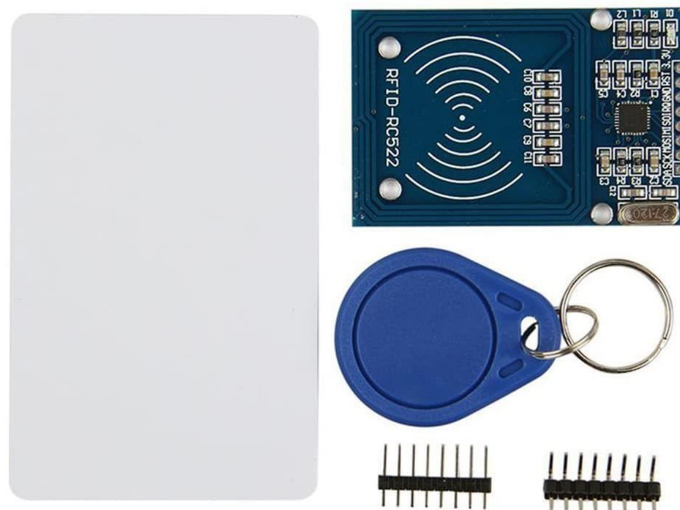


Fig. 2.17: RC522 RFID Kit with Blank Card and Key [22]

The RFID keychain contains 1kb of data and can be programmed to have a unique ID identifiable by the RFID reader module [21].

Buck Converters

Several of the components require a 5V and 3.3V power supply. The 3.3V devices could easily be powered by the Raspberry Pi 3.3V pin, but a buck converter was required to convert the 12V to 5V. There are two buck converters used in this project, one is an 8A USB buck converter shown in Figure 2.18, and the other is a 5A buck converter shown in figure 2.19.

Fig. 2.18: 12V to 5V 8A USB Buck Converter [23]

Fig. 2.19: DROK 12V to 5V 5A Buck Converter [24]

The Raspberry Pi and wireless charging pads use the USB buck converter. The other buck converter is used to power the LED lights, GPS module, and servo.

Switching MOSFET

Since the user has the option to turn feature on and off, MOSFETs are used as switching devices. The speakers, the GPS module, the LED lights, and the servo for the locking

mechanism use FQP30N06 N-Channel Power MOSFETs. These MOSFETS were selected because they are low logic level MOSFETs that can be activated with a 3.3V GPIO pin from the Raspberry Pi. The MOSFET is a 60V 32A device, which exceeds the required power needed for the 12V and 5V devices.

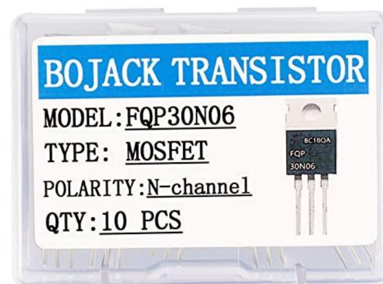


Fig. 2.20: FQP30N06 N-Channel Power MOSFET [25]

2.3.2 The Lid

The lid will feature the touch screen for user interaction and wireless charging. The lid must provide ample room for the 7" touch screen, as well as cup holders.

Wireless Charging

Another very important requirement of the Smart Cooler is the wireless charging pad. The Qi wireless charging board made by Adafruit was selected for the project. It has a charging distance of 2-8 mm which should easily be able to transmit through the plastic of the lid. Between this and the Wired USB Charger the user will be provided with charging options that should cover most mobile devices. The wireless charging module outputs max 1 A at 5 V, so ideal for smartphones. Two of them are recessed into the lid.

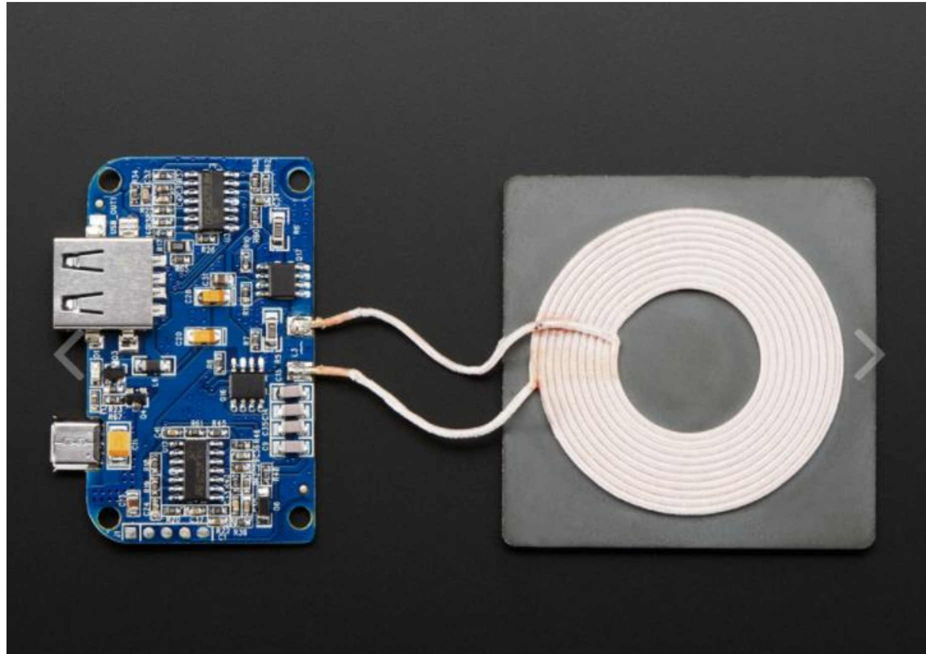


Fig. 2.21: Qi Wireless Charging Transmitter [26]

Touch Screen

The touch screen is very important as it is designed to be the primary means of interfacing with the system. This touch screen also supports inputs for up to 5 fingers at once, making it ideal as the main user interface. The touch screen uses USB for the touch inputs, and HDMI to display the Raspberry Pi monitor screen.

Fig. 2.22: SunFounder 7” Touch Screen [27]

Lid Position Sensor

For the locking mechanism to activate, the system needs to know when the lid is closed. A reed switch accomplishes this as it is a magnetic switch that closes when the magnet gets near the reed switch.

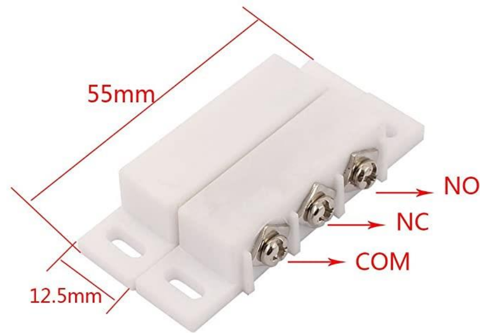


Fig. 2.23: Reed Switch [28]

2.4 Power Budget

The power budget shows how much power will be required from the batteries at any given time. The wattage was derived using the power equation.

Power Equation $$P = I \cdot V \tag{2.2}$$

Table 2.1 shows the power budget, with a total required amp of 15.1A. The batteries selected have a maximum discharge current of 42A.

Table 2.1
Power Budget

Component	Model	Voltage	Amp	Watts
Microcontroller	Raspberry Pi 4	5V	3A	15W
Temperature Sensor	DS18B20	5V	1mA	5mW
Wireless Charging	Qi Wireless Charging Transmitter	5V	1A x 2 = 2A	10W
Wired USB Charging	YM1236 Dual USB Charger	5V	2.1A	10.5W
Solar Panel	Eco-Worthy	12V	0.833A	10W
Servo	Micro SG90	5V	360mA	1.8W
RFID	RC522	3.3V	26mA	85.8mW
GPS Module	BN-880	5V	50mA	250mW
Touch Screen	Sunfounder 7”	5V	480mA	2.4W

Amp Board	DAMGOO	5-27V	>3A	Up to 200W
LED Light Strip	Alitove WS2812B RGB LEDs	5V	4.05A	20.25W
Analog to Digital Converter	MCP3008	5V	500uA	2.5mW
Total			15.1A	260.3W

Chapter 3

Contribution

Summary

The process for building the cooler was documented in this chapter. The design integration and implementation go into detail of how the cooler was assembled and what the cooler was designed to do. Any issues encountered during the build and final results are mentioned. The success and failure rates of the project are also in depth.

3.1 Introduction
3.2 Design Integration and Implementation
3.3 Design Troubleshooting and Results
3.4 Discussion of Success and Failure Rates

3.1 Introduction

This chapter will discuss the contributions of the group members as the project was completed. It will detail the design and implementation of ideas, and the integration of said ideas as the project improved through iteration after iteration.

3.2 Design Integration and Implementation

This section discusses the implementation and design process of the project in detail. It will give detailed breakdowns of the design and integration process for the modules discussed and presented in chapter 2.

3.2.1 Power System

The system is meant to be a portable power source. It runs on 12V rechargeable batteries that can be charged through wall outlet, car port, or solar panel. The output of the batteries provides 12V power to necessary devices, while buck converters are used to convert the 12V power to 5V.

12V Rechargeable Batteries

The system is designed to be powered by a pair of 12V rechargeable lithium batteries that are going to be set up in parallel. Initially, it was decided to go with the $LiFePO_4$ that were rated for 26Ah, but due to availability issues batteries with less capacity were bought. The batteries used are 16Ah batteries. Regardless, they still provide ample power needed to run the system. The batteries are a significant amount of the weight in the system, with that in mind it was decided to place the batteries in the back of the cooler over the wheels.

Fig. 3.1: 12V 16Ah Batteries

Power Distribution

With the system dealing with so many voltages internally a system had to be designed to properly control and distribute them. Terminal blocks were selected to be the backbone of the power distribution and should provide a good and simple way to route the different voltages that will be created from the different charging circuits.

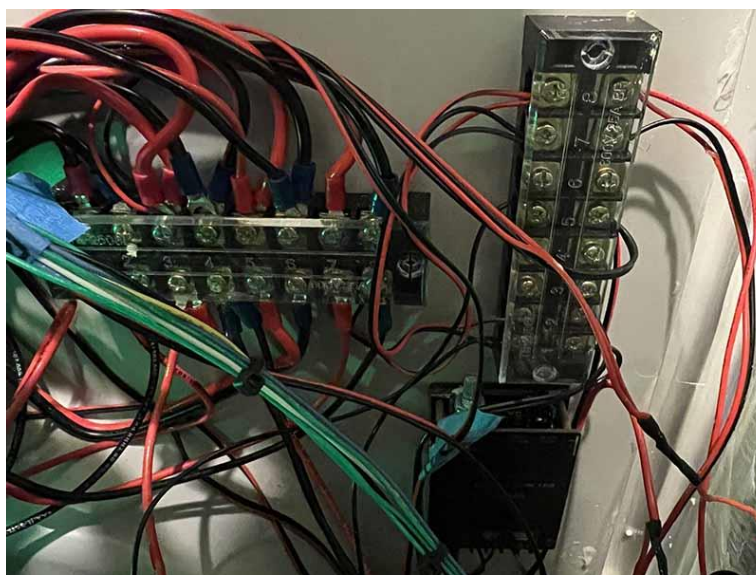


Fig. 3.2: Terminal Blocks

Battery Charging

Rechargeable lithium batteries require special current regulation to properly charge, so to meet this demand several consumer products had to be bought and modified. These

products will facilitate three types of charging options the system which will give the system charging options which should cover nearly all cases where the Smart Cooler is being used.

Wall Mounted 120VAC

The main charging option for the system is going to be the 120VAC charging circuit that is the standard voltage option in the United States. To handle this the NOCO Genius 10 battery charger was selected. This is a consumer product and for it to meet the needs of this project and be mobile modifications had to be made.

Fig. 3.3: Lithium Battery Charging Device

The charger did come with a mounting bracket that was used to mount it to the interior wall of the cooler. The charger has two sets of cables connected to it, one that plugs into the wall outlet and the other which supplies the 12VDC to the batteries for charging. This 12VDC will be used in several other parts of the system thanks to the terminal blocks.

Solar Panel

The solar panel used provides a 12V 1.5A output. It wasn't selected to completely power the system, but to just give a trickle charge while the cooler is being used outside. After doing some battery tests it was determined that the cooler power is not as demanding as anticipated and even without the solar panel the cooler could last over a week outside.

Fig. 3.4: Solar Panel

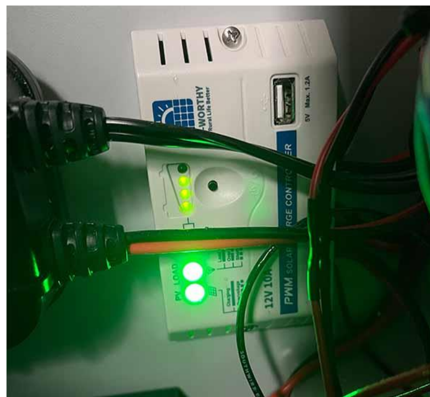


Fig. 3.5: Solar Panel Charging Module

Car Port Charging

To facilitate the mobile charging from a 12VDC Voltage a cable was bought, and a port was installed into the cooler that was connected to the terminal box. This connection was tied into the inputs to the solar charging panel circuit. This circuit handles the 12VDC generated by the Solar Panel so by routing the car port voltage through it this provides the charging function and the current adjustment needed for to properly charge the batteries in maintained.

12V to 5V Buck Converters

The system requires 5VDC in several places to operate and initially buck converters were used to handle the voltage conversion. During testing and implementation, it became obvious that the Raspberry Pi and Wireless charging modules required significantly higher current then the rest of the systems and to meet that demand a buck converter was selected that supplied a much higher current and that also distributed it via USB C which

also happened to be convenient for the connections to our modules. To handle the other 5VDC a buck converter was used that was rated for 5Amp and that more than exceeded the voltage required for the other modules of the system.

3.2.2 Device Charging

The Smart Cooler will offer several options for device charging to allow the user to charge most modern mobile device with the battery system. At time of design that this will require wired and wireless charging options.

Wired Charging

The wired charging functionality will be covered by the dual port car charger. This charger supplies ports for USB 2.0 and USB type C. These ports will cover most cable connections for mobile devices. This port was installed and hard mounted into the shell of the cooler by drilling a hole through the insulation in a convenient location.

Fig. 3.6: Running Wires Through Interior Panel

With the location set for the port, because of the design of the power distribution system, powering the USB port became just a matter of wiring it into the terminal block on the 12VDC terminal. The figure below shows the interior of the cooler and the wires from the back of the port to the terminal block.

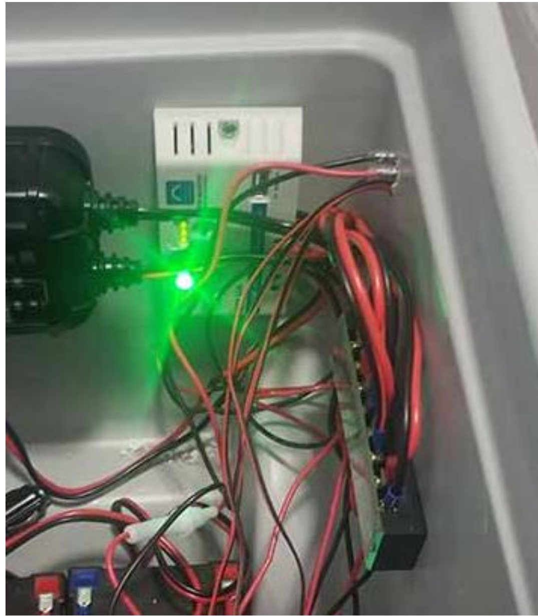


Fig. 3.7: Wires Run Through Wall

With the port properly wired it was then secured in place with waterproof two-part epoxy. The following figures are of the final location of the port and the port illuminated during operation.

Fig. 3.8: Wired Charging Port

Wireless Charging

The wireless charging function will cover the charging options for mobile devices not supported by the wired function. The function will be covered by using two wireless charging modules.

Fig. 3.9: Wireless Charging Pads in Lid

The largest consideration with the wireless charging modules is the significant current draw needed to power the modules. This was also an issue that we had regarding the Raspberry Pi because it also required rather high amperage to be supplied over USB. Thankfully the buck converter that was selected for the needs of the Raspberry Pi has enough ports to support the wireless charging modules.

3.2.3 Controller

The entire system uses a Raspberry Pi 4 as its microcontroller. The Raspberry Pi either controls the electronic devices directly, as components run on 3.3V or 5V, or through GPIO switch that uses a MOSFET to control 12V devices. As the Raspberry Pi provides power for 3.3V and 5V components, the 3.3V power pin was used to power the analog to digital converter used for the light sensor and battery level indicator, and the RFID. The 5V power devices, such as the GPS, LED lights, and servo, were powered by a separate 12V to 5V buck converter, as those components would require more amps than the Raspberry Pi could safely provide. In total, 15 GPIO pins were used, as well as the 3.3V and ground pins. The following table below shows the pinout, which devices use which pins on the Pi.

Table 3.1
Raspberry Pi Pinout

Component	Pin Name	Pin Number	Pin Type
MCP3008 (A/D Converter) **3.3V**	CLK	23	SLCK
	DOUT	21	MISO
	DIN	19	MOSI
	CS/SHDN	26	GPIO-07
RC522 (RFID) **3.3V**	SDA	24	GPIO-08
	SCK	23	SCK
	MOSI	19	MOSI
	MISO	21	MISO
	RST	22	GPIO-25
BN-880 GPS Module **5V**	SDA		
	TX	10	Pi RX
	RX	8	Pi TX
	SCL		
	Power	40	GPIO-21
Temp Sensors	Data	7	GPIO-4
Magnet Sensor		27	GPIO-0
LED Lights	DIN	12	GPIO-18
Servo **5V**	Power	36	GPIO-16
	DIN	13	GPIO-27
Bluetooth Speakers	Power	38	GPIO-20

3.2.4 LED Lights

The LED light strip was measured and cut to have exactly 73 lights that circle around the rim of the cooler. The light strip is a 5V powered strip, with each of the LEDs being RGB. The light strip gets its power from the 12V to 5V buck converter. The color of the LEDs can be programmatically changed by the Raspberry Pi because the strip has a data

pin. The colors can be changed to be either red, blue, white, or rainbow. Other colors can be attained, as the values are adjusted by setting the red, green or blue values from 0 to 255, 0 being no brightness and 255 being full brightness.

Fig. 3.10: Blue LEDs (Top Left), Rainbow LEDs (Top Right), Red LEDs (Bottom Left), White LEDs (Bottom Right)

In order to programmatically change the colors of the LEDs, a Python library was used called rpi_ws281x. This library is used specifically for WS2812B LED strips. With this, the colors can be adjusted to do different effects, but in this instance, they were only used to light up as solid colors.

The lights have the option of being able to be turned on or off on the touchscreen. They also have the option of being set into auto or manual mode. In manual mode, the LEDs can be turned on or off, but only when the lid is open. The lid uses a reed magnet to detect if it is open or closed. This way, the LED lights will not be on when the lid is closed, possibly draining battery. In auto mode, the LED strip uses a photosensor to detect the light levels outside, and if the light is detected at a certain threshold, the LEDs will turn on. The photosensor is encased in a clear plastic case that has been epoxied onto the cooler to make it waterproof.

Fig. 3.11: Photosensor Mounted on Exterior of Cooler

3.2.5 GPS

The GPS module uses 5V power, which it received from the 12V to 5V buck converter. The GPS module connects the TX and RX pins connected to the RX and TX pins on the Raspberry Pi, respectively. A bracket was 3D printed for the module, as the cooler is expected to be transported over rough terrain, so the bracket keeps it from being knocked out of place.

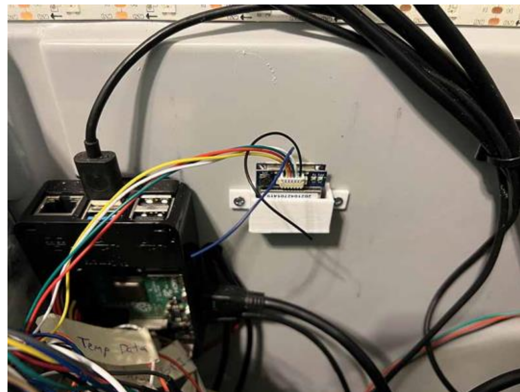


Fig. 3.12: GPS Mounting Bracket

The library used to get the GPS module working is the gpsd library. The GPS module can be enabled and disabled by the touchscreen or mobile app. When the GPS is enabled, it received coordinates, and when it is disabled, it does not receive coordinates. Though the location cannot be seen on the touchscreen, the coordinates are sent to the mobile app using Bluetooth. If the GPS module is enabled, it will send the last known coordinates of the cooler. This allows for the coordinates of the cooler to be saved on the phone, so that when the user is out of range of the cooler, the last location will still be known. The

location can then be viewed on the mobile app using Google Maps. Google Maps can then provide directions to the location.

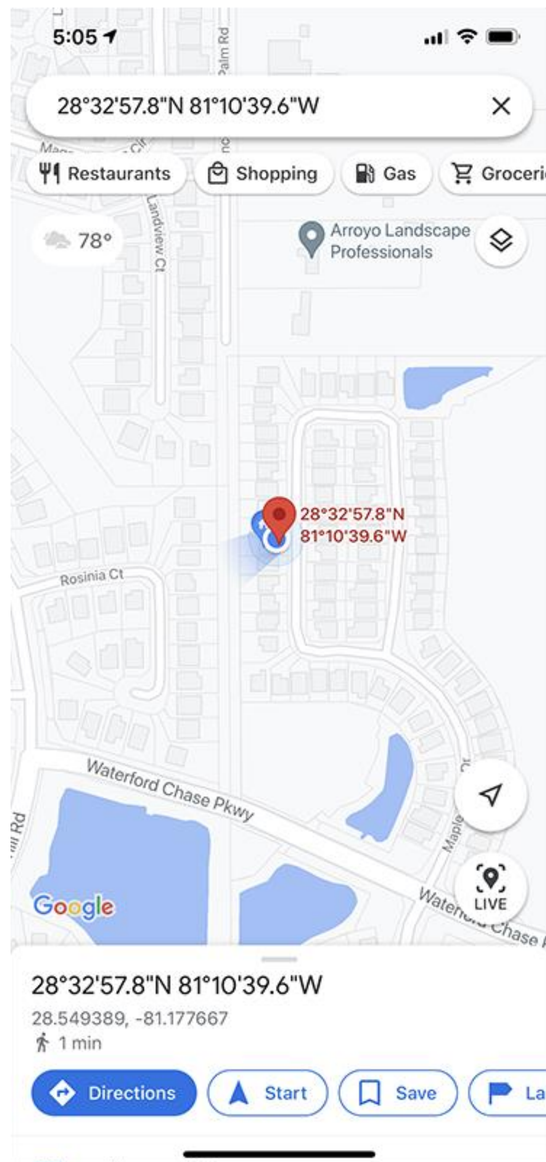


Figure 3.13: Google Maps Using GPS Coordinates

3.2.6 Locking Mechanism

The locking mechanism feature allows the cooler lid to be locked and unlocked. The lock can be activated either by using the touchscreen, the mobile app, or using an RFID tag. The locking system contains a door position sensor shown on the left and the locking component shown on the right of the figure below.

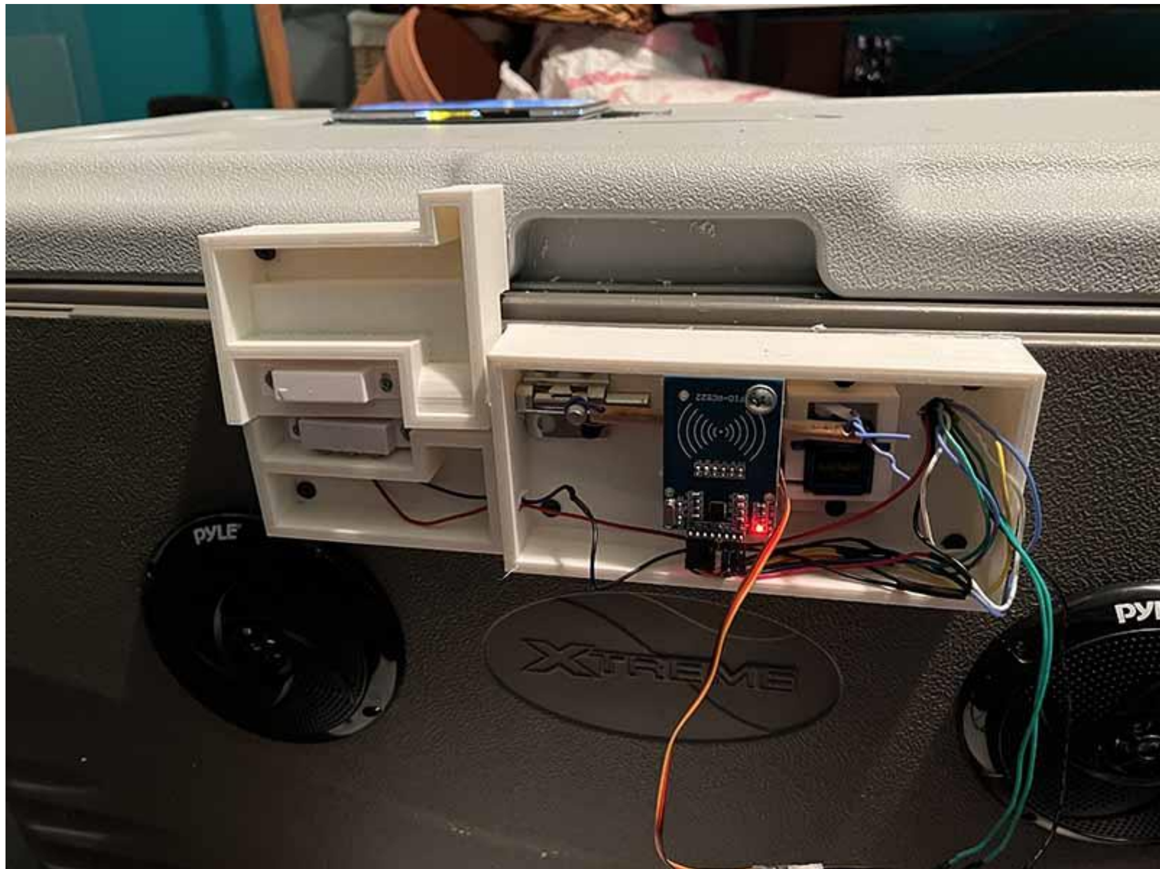


Figure 3.14: Locking Mechanism without Covers

The door position sensor uses a magnetic reed switch wired in the normally open position which connects to a Raspberry Pi GPIO. The wiring is simply a 3.3V connection going to the reed switch, which then goes to the GPIO. The GPIO is set as an input so that when the reed switch is closed, a voltage is detected at the GPIO pin, telling the Raspberry Pi that the lid is closed.

The locking component contains an SG90 servo, a wooden dowel, and a metal sliding latch. The servo is programmed to rotate only 90 degrees, as the rotational motion is converted to linear motion with the dowel, creating the exact distance to lock. The main component is the servo, as that is what is being controlled when the touchscreen or mobile app tells the sliding lock to activate. The RFID is another form of control. By simply tapping a RFID tag against the reader, the locking will activate.

The locking device can also be set to auto by going to the settings menu. In the auto feature, as soon as the lid is closed, the lock will activate. This is simply for added convenience. Some logic needed to be added to allow a user to unlock the lid, since having the lid closed will always lock the lid. Therefore, whenever the lid needs to be

unlocked, a three second delay is added to allow the user enough time to open the lid. After the three seconds passes, the lid will lock again.

3.2.7 Bluetooth Speaker Entertainment System

The Bluetooth Entertainment system consists of marine speakers, the Bluetooth amplifier, and the Raspberry Pi. Marine Speakers were selected because like everything else in this project it is expected to operate around water and be able to operate after being splashed. To recess the speakers in the cooler, much like the wired charging plug, holes were cut and drilled in the cooler shell.

Fig. 3.15: Holes for Speakers

Powering the speakers would typically come from the amplifier that the speakers are connected to, however the speakers that were selected have LEDs in them that requires power.

Fig. 3.16: Bluetooth Speakers Turned On

To control the operation of the speakers the amplifier's operation is controlled by the Raspberry Pi.

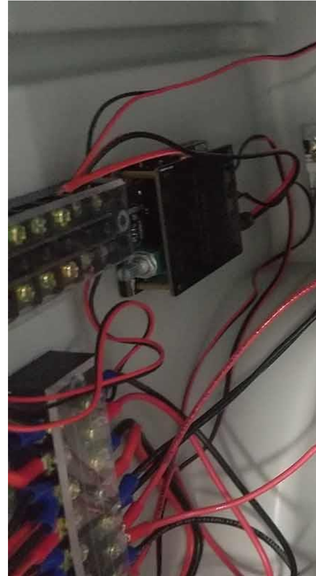


Fig. 3.17: Speaker Amp Board

3.2.8 Touchscreen

The touchscreen used in this project is a 7" display Raspberry Pi compatible touchscreen. It uses a USB cable to detect input and an HDMI cable, both connections going to the Raspberry Pi. On startup, the Raspberry Pi loads the program that displays a GUI interface on the screen. The GUI was created using a library called guizero. The GUI, as well as the mobile app, are direct inputs that allows the user to control the cooler. The GUI on startup is shown in the figure below.

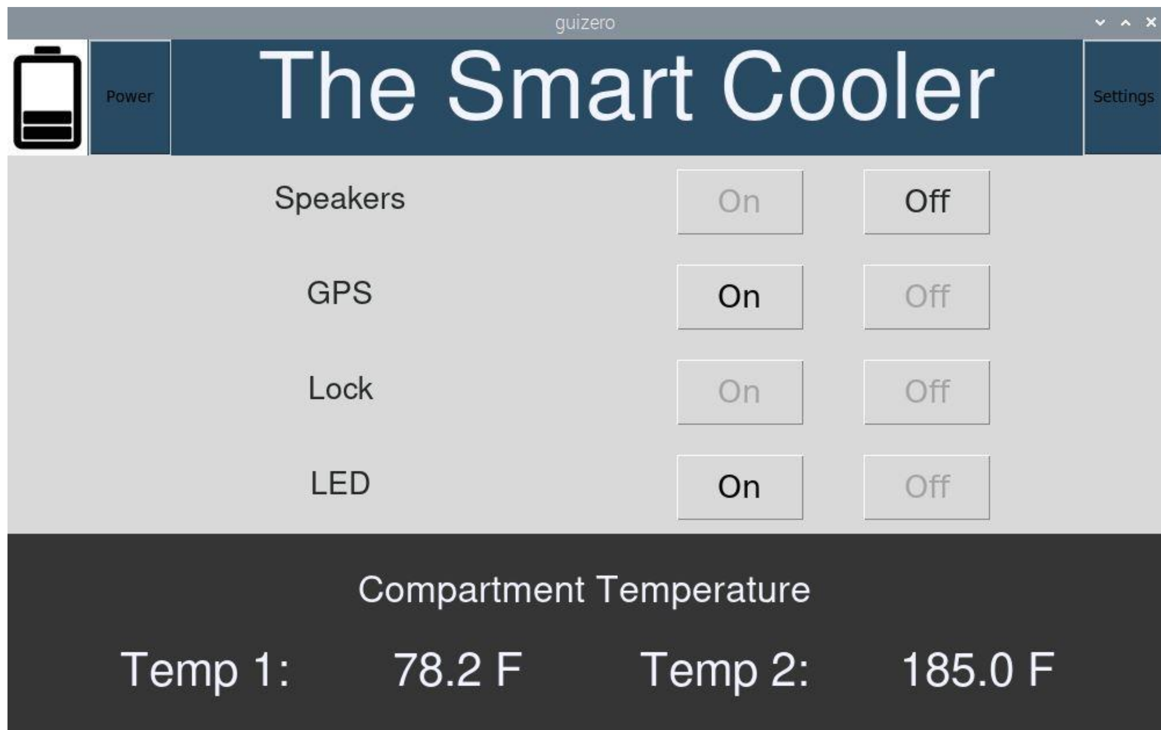


Fig. 3.18: Touchscreen GUI Main Screen

The GUI main screen features several controls. On the top left are the battery level indicator and power button. By pressing the power button, the Raspberry Pi will save all current settings before shutting down so that on startup the cooler will be exactly as it was set before shutdown. It achieves this by saving a text file on the operating system containing all the variables of the program and setting the variables on bootup. The battery level indicator features five bars, from dead battery to full. After draining the batteries, it was established that when the battery reaches below 12V, the system cannot operate, and when the battery reaches over 14V, the system is fully charged. The battery level indicator works by using a simple voltage divider and an analog-to-digital converter. The 12V is converted to 3.3V using resistors, and a range from 0 to 3.3V allows the system to determine the voltage level since its proportional to the battery voltage.

The middle section contains control buttons for the features. The speakers can be turned on and off. This is attained by splitting the speaker ground connection using a power MOSFET that is controlled by a Raspberry Pi GPIO. The GPS is also controlled the same way. When the GPS is turned off, the last known coordinates are still saved onto the cooler. The locking feature is also controlled on the touchscreen. In the figure above, the locking buttons are disabled because the lid is open. It doesn't make sense to lock or unlock the lid when it is already open, so the buttons are disabled for intuitiveness. The LED lights also

contain the same feature but opposite of the lock. When the lid is closed, the LED buttons are disabled as there is no point of having the LED lights on when the lid is closed. The bottom values are the temperature measurements for compartments one and two.

The settings button on the top right of the GUI takes the user to the settings screen, where the system functionality can be modified a little more advanced.

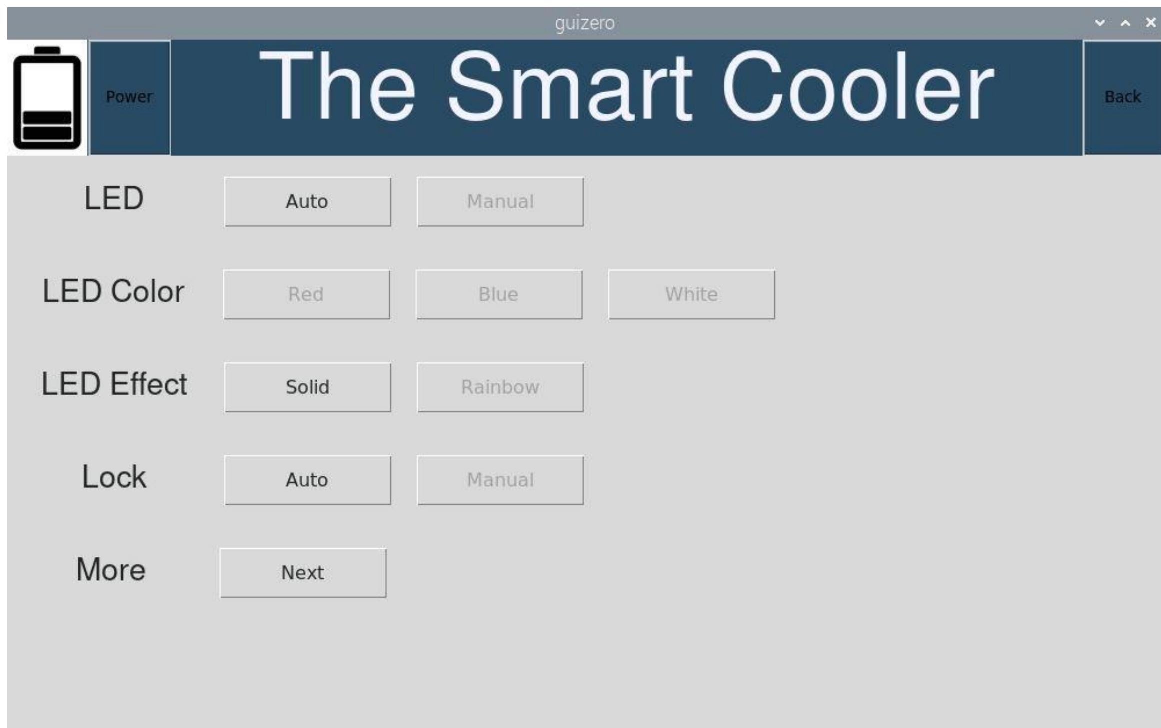


Fig. 3.19: Touchscreen GUI Settings Screen One

Going down the list, the LED lights can be set to auto or manual mode. Auto activates the lights when it is dark outside. The LED light colors can be changed to red, blue or white when in solid mode. When the color effect is set to rainbow, the solid color buttons are disabled. The lock can be set to auto or manual, where auto will lock the cooler when the lid is closed automatically. Tapping the “Next” button takes the user to the second settings screen with more editable features.

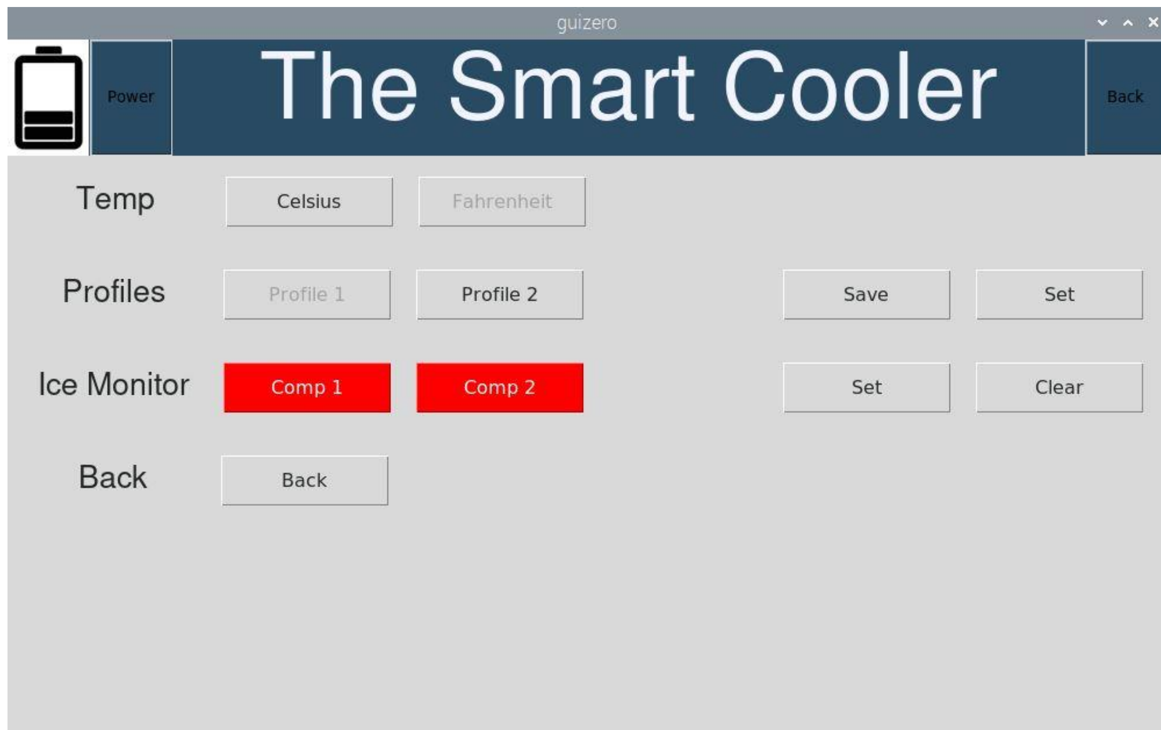


Fig. 3.20: Touchscreen GUI Settings Screen Two

Here, the user can switch the temperature readings to Celsius or Fahrenheit. Profiles is a way for different users to save their preferred settings. The cooler can save up to two profiles. If for instance someone borrows the cooler and changes the settings, the owner can easily revert back to the original settings by selecting their profile and tapping “Set”. Basically the profiles save the program variables to an external text file similar to when it shuts down. Next, the ice monitor setting allows the user to get a notification when the ice has melted and the temperature is high in the compartment. In Figure 3.20, both compartment buttons are red because they are not activated yet. First, the user must tap the compartment they would like to monitor, then hit “Set”. The activated compartment will turn green, as shown below in Figure 3.21. Both compartments can be activated at the same time.

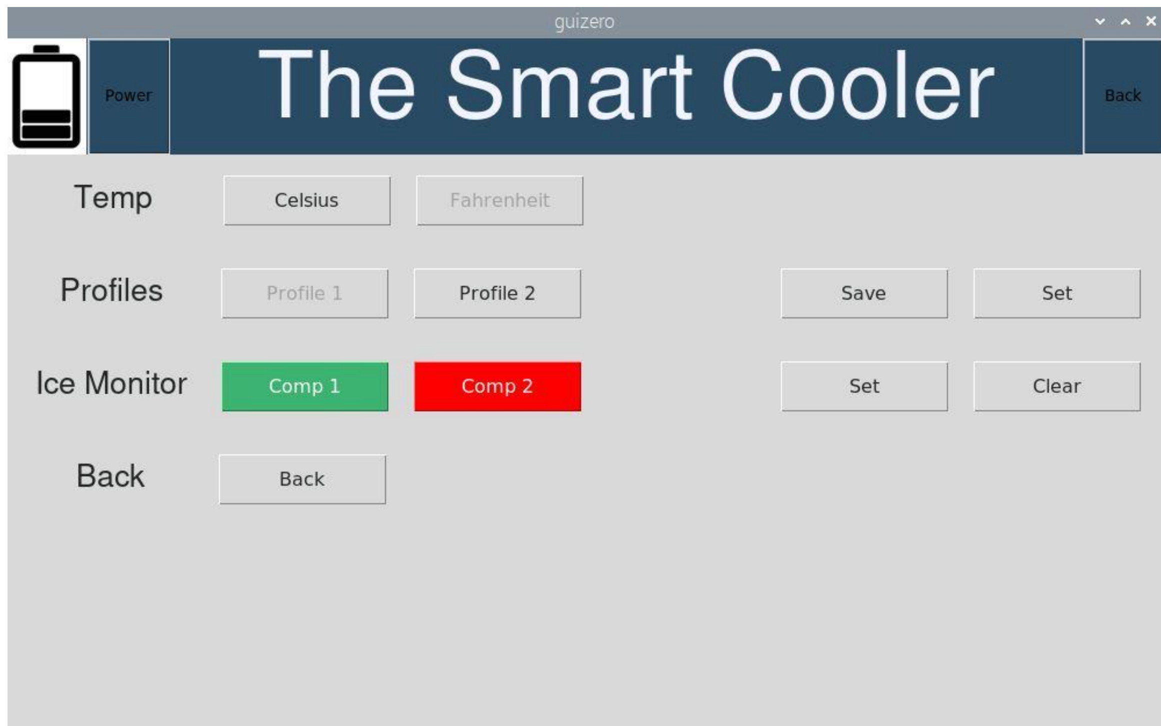


Fig. 3.21: Touchscreen GUI Ice Monitor Compartment One Activated

Once the compartment goes above 90 degrees Fahrenheit, a notification will display on the screen with a message telling the user that the compartment ice needs to be replaced.

Fig. 3.22: Touchscreen GUI Ice Replacement Notification

3.2.9 Mobile App

The mobile app initializes with a loading screen where the app is scanning for Bluetooth devices. The screen has an animated cycle showing that scanning is in progress.

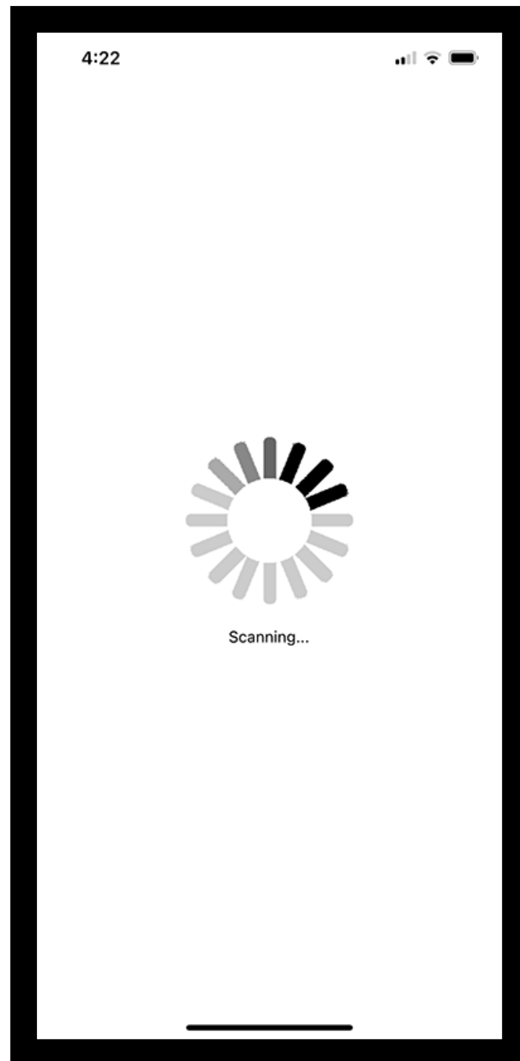


Fig. 3.23: Mobile App Scanning Screen

Once the app has identified a Smart Cooler device, the screen then changes to ask the user if they would like to connect to the device.

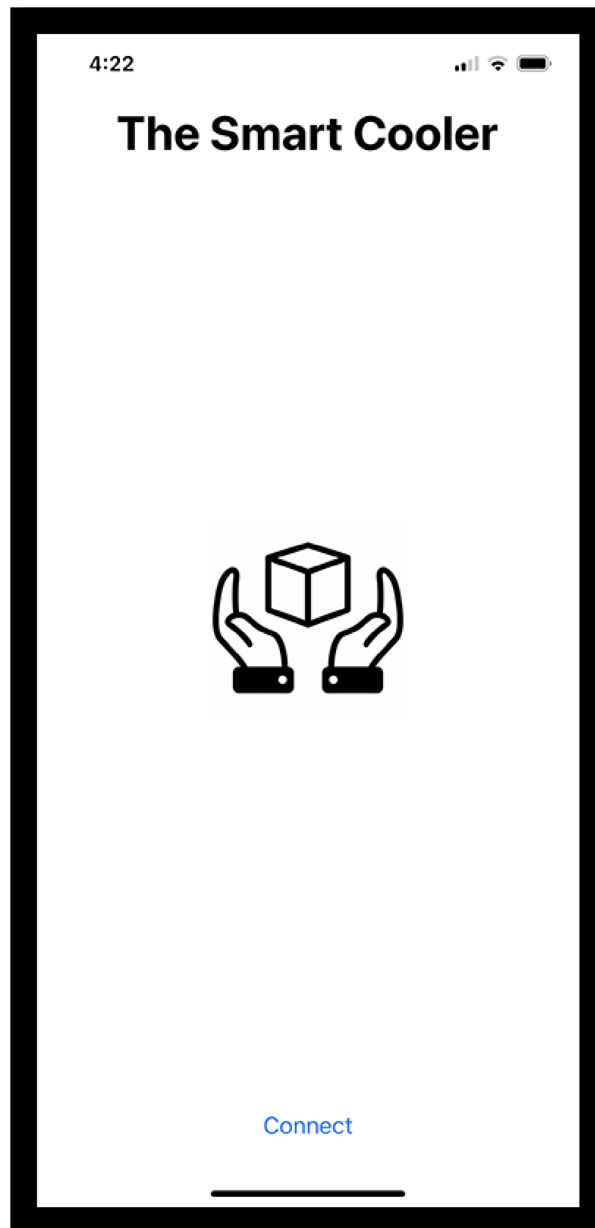


Fig. 3.24: Mobile App Connect Screen

Once connected, the mobile app will start receiving data from the Raspberry Pi. The figure below shows the options on the main screen. The top left shows a cogwheel which is a settings button that will load the settings screen. The battery image on the top right shows the battery level. The speakers, GPS, lock, and LED lights can be powered with the slider buttons, and then the temperatures for both compartments and the GPS location are shown below.

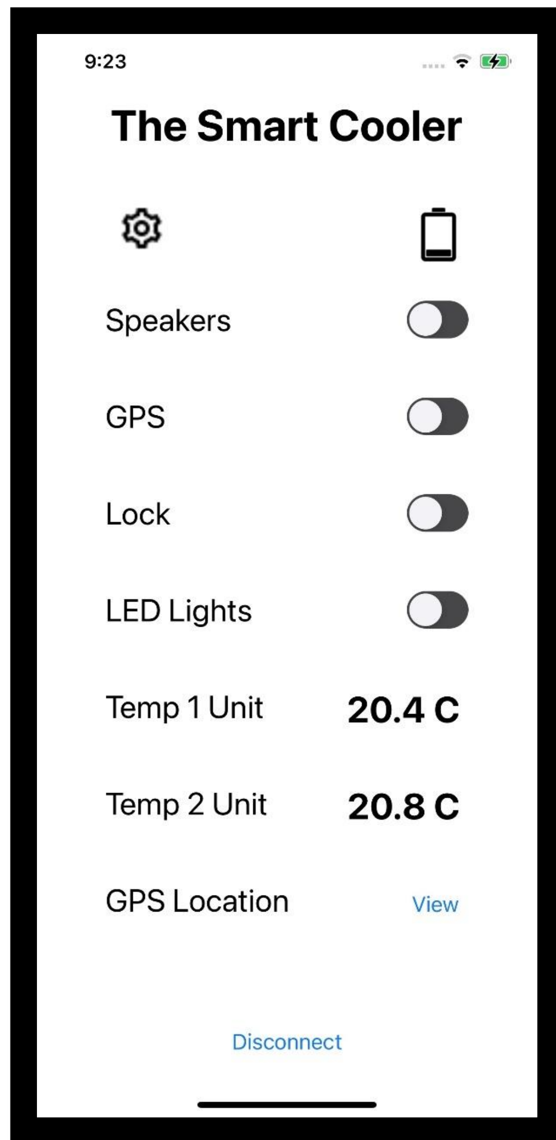


Fig. 3.25: Mobile App Main Screen

When the “View” text is pressed for the GPS location, the app uses Google maps to show the cooler location. The Raspberry Pi is emitting GPS coordinates, so the coordinates that get entered into Google maps are latitude and longitude coordinates.

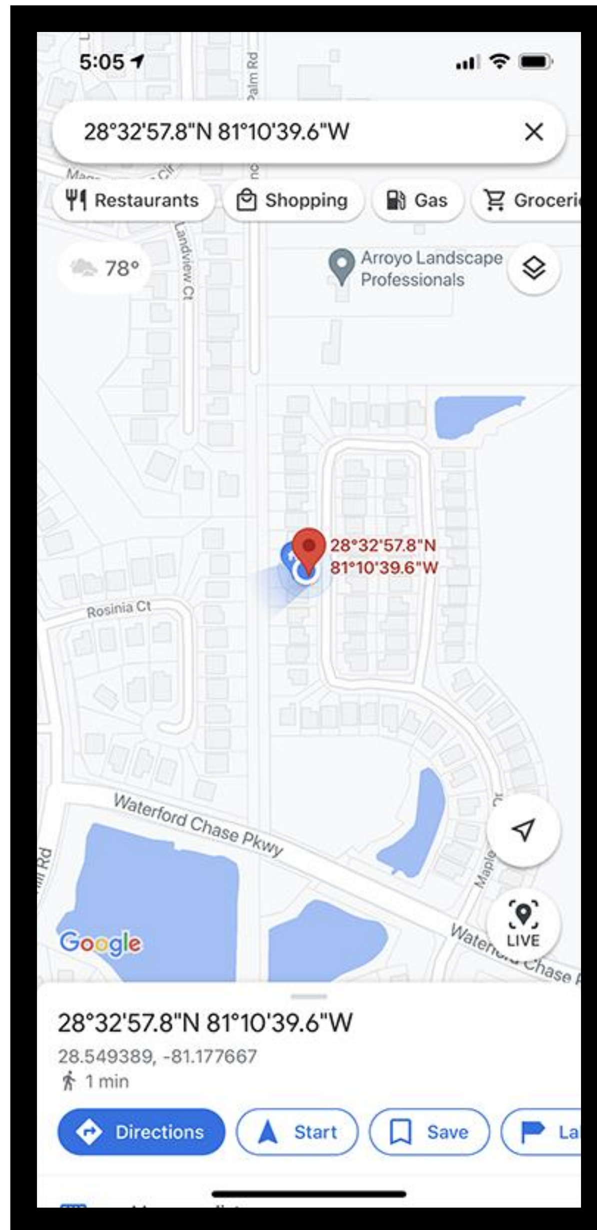


Fig. 3.26: Mobile App GPS Location Screen

The settings screen has several customizable features shown in the figure below. Basically everything that can be changed on the touchscreen can also be changed on the mobile app. Note that the user cannot set or save profiles, as well as enable ice notifications on the mobile app. That would have to be done on the touchscreen only.

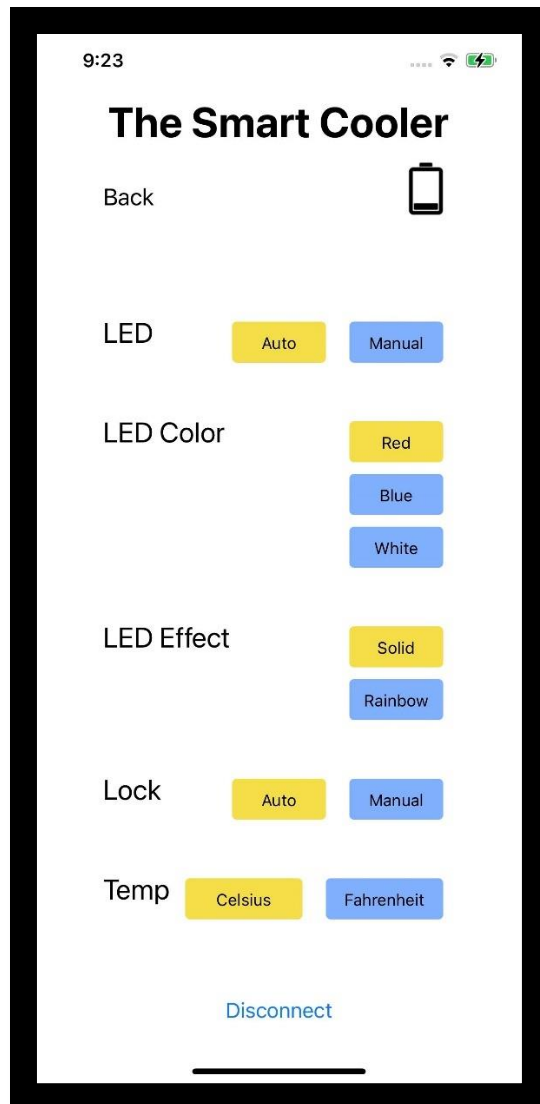


Fig. 3.27: Mobile App Settings Screen

If ice notification are enabled for any compartment, the mobile app will notify the user by displaying a button on the top for the compartment that requires ice.

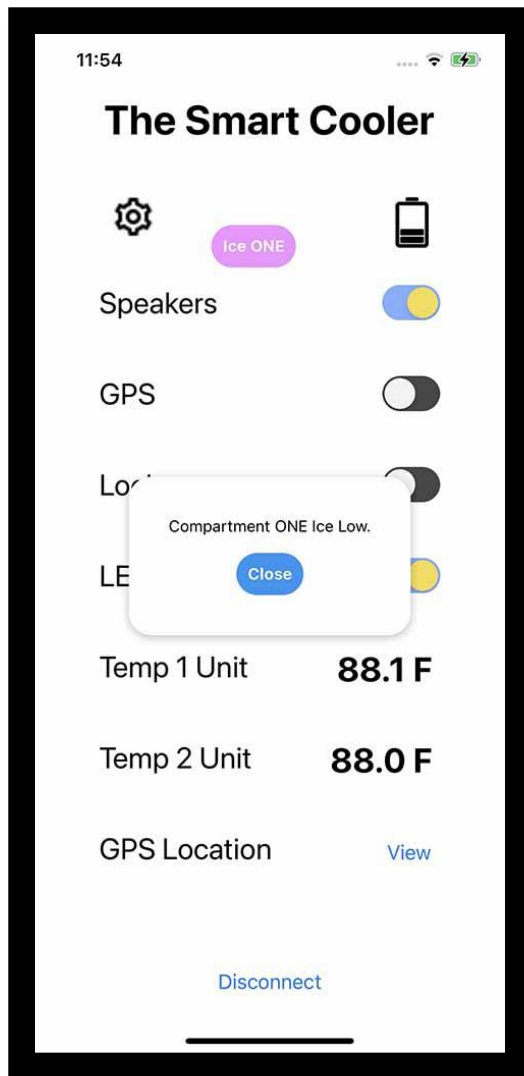


Fig. 3.28: Mobile App Ice Notification Compartment One

The mobile app was created using React Native. The app was deployed on both an iPhone 11 and Galaxy S9. To incorporate Bluetooth into the app, the “react-native-ble-plx” library needed to be implemented. Also, to develop for an iPhone, an apple MacBook must be used to deploy.

3.2.10 Software

There are several scripts that were written for this project. The Raspberry Pi required a script that would start on bootup, and well and the main functioning script. The mobile app also required its own script programming. The Raspberry Pi flowchart can be seen in Appendix D and software code can be seen in Appendix E. The mobile app code can be seen in Appendix F.

Raspberry Pi Software

The Raspberry Pi uses the Python programming language, so all the scripts and libraries used were in Python. Initially, the Raspberry Pi has a script that starts on bootup. The tasks that get complete on startup is that the necessary serial communication socket get enabled for the GPS to read the coordinates. The next part is to execute the actual main program.

Once the main program gets executed, the first thing that happens is the program looks for a file that has stored the setting variables. When the cooler is powered off, it saves the current settings into a text file stored on the Raspberry Pi. On bootup, the program initializes the main variables for the program uses that text file so that the cooler will contain the same settings it had before it shut down, instead of resetting every time it is powered up.

Once the initial variables are set, the program loads up the touchscreen GUI. The GUI uses a Python library called guizero. The library simplified the process of adding text, images, buttons and screens to the project. The library uses a loop to detect user activity and is always set as the last loop initialized in the program, per the documentation. The GUI takes care of the front-end logic. What happens behind the scenes is much more.

Since Bluetooth needed to be used, the program sets up the Raspberry Pi as a Bluetooth Low Energy (BLE) device. The Bluetooth service uses the dbus service loop to send and receive information. Note that the GUI requires a loop, while the Bluetooth service requires another loop. To run two loops, and the other loops mentioned later, Python threading was used. This will be discussed in more detail in section 3.3. For Bluetooth to advertise to other devices, a Bluetooth service needed to be initiated. The service contains the characteristics, or the variables, that will be read and written to. The service and characteristics require UUIDs, which are basically like serial numbers. The program has 15 characteristics being advertised. They include the Bluetooth speakers power, the LED lights power, the lock enable, the GPS power, compartment one temperature, compartment two temperature, GPS location reading, LED mode setting, LED color, LED effect, lock mode setting, temp units setting, battery level reading, compartment one ice notification, and compartment two ice notification. All of these characteristics either send the state of a variable to a requesting device or writes over the variable from the device.

There are five loops total, one for the GUI, Bluetooth, RFID, LEDs, and the main loop. The RFID loop simply waits for an RFID tag to be read by the RFID reader and will execute. Once done executing, it will wait for another tag reading. The LED lights also needed to be one its own load for the library that runs the LED lights.

The main loop is used to read sensor values and set the variables accordingly. The loop first reads the light sensor value. If the LED lights are set to auto, the sensor value is used to detect if it is dark outside and activate the LED lights if it is. Next the battery level is also read. Both the light sensor and battery level use the analog to digital converter to read values. Then the temperature sensor values are read in Celsius. If the temperature units are changed in the settings, then the temperature values will be in Fahrenheit. These temperature readings are used for the ice notification system. If the temperature is above a certain point, then the notification will activate. If the lock is set to auto, then the lid is checked to see if it is closed, and if so, lock the lid. If the GPS module is on, the GPS coordinates will be checked. The loop then sleeps for 1.5ms to allow itself time to finish executing all tasks before restarting.

Mobile App Software

The mobile app uses React Native to deploy to both Android and iPhones. The library uses JavaScript, which is a popular web browser-based programming language. The main function of the mobile app is to communicate with the Raspberry Pi using Bluetooth. When the app is first launched, the app does a scan for Bluetooth devices. It continues to scan until it finds a device names "The Smart Cooler", and then tries to connect. On first time connecting, an authorization will be prompted, and by clicking "OK" it should authorize. Then the connect screen shows. By pressing connect, the mobile app and Raspberry Pi are now able to send and receive data to each other.

React Native works by using components. Each component can have its own state, and once that state changes, the component re-renders. Once connected, the program reads the different characteristics available from the Raspberry Pi. These characteristics each become their own component, and the variables is what becomes the state of the component. So, say for example the LED lights button is set to "OFF", so the component state for the button will be off, and the button will be rendered off. If someone were to press the LED lights button on the touchscreen on, the Bluetooth would send a variable set to "ON", and the app will change the state of the LED lights component to on and re-

render the button to be on. This is essentially happening to any of the buttons on the mobile app. Aside from the buttons, there are the temperature readings, the battery level, and GPS coordinates which can only read the value from the Raspberry Pi but not modify it. The GPS is simply a button that takes the coordinates and uses a google maps link. The mobile app can tell the difference between characteristics by the UUID.

3.3 Design Troubleshooting and Results

Any issues that were encountered during the process are discussed in this section. Though minor issues aren't mentioned like wires not being in the right place or parts suddenly not working from being accidently disconnected, they have occurred. The final results of the project are also shown.

3.3.1 Troubleshooting

A lot of 3d printing needed to be done for the project to create custom parts. First was making sure the printer was set to the correct settings. The program used to slice the models was Ultimaker Cura. Because some of the models required printing surfaces that were above the build plate, the program needed to be modified to allow printing support beams. For instance, the sliding lock that was printed needed to have a hollow case, so the program filled in the gap with support beams.

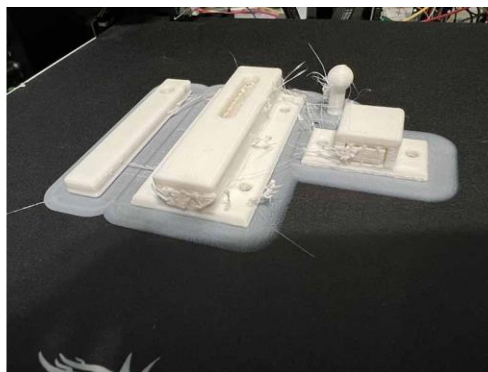


Fig. 3.29: 3D Printed Sliding Lock

However, looking at the print, the support beams made the part unusable, unless the support beams were taken out by cutting and sanding. The hassle was avoided by just buying metal latching locks. Also, the plastic strings were extruded out of the main body, making the print look sloppy.

Another issue was with the main program. Unfortunately for the program, several loops needed to be run at once. Bluetooth needed to use a loop, the touchscreen GUI needed a loop, the LED light strip had its own loop, the RFID required its own loop to wait for an RFID device to activate, and a separate loop was run for all the sensor data. Unfortunately, not all these interfaces could be run on the same loop, as some wait for a response, which would halt the program. To fix this, Python threading was used. Threading allows the users to run code concurrently. "A thread is a separate flow of execution. This means that your program will have two things happening at once. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to" [29]. Initially there were more than five threads, however, issues started to occur where some threads weren't running, so the number was reduced.

Another issue was with setting the LED lights to auto. When the LED lights are set to auto, the Raspberry Pi reads the value of the photosensor to determine if it is dark outside. Initially there were issues as the light from the cooler was affecting the sensor value. Since the LED lights were so bright, when the lights are activated, the light reflects out of the cooler and radiates to make its surrounding area more visible, which would then cause the photosensor to tell the Raspberry Pi it was light outside. This caused a perpetual on and off cycle where the cooler is placed outside in the dark, the light would turn on, and since the photosensor was reading light values it would turn off the lights. The lights would then turn back on as it was dark outside, and the cycle would continue. To fix this, a variable was added in the program to save its state, where once the lid is open and it is dark outside, turn and keep the lights on until the lid is closed.

For the locking mechanism, there were three devices that needed to share a ground. The servo, the RFID, and the door position sensor all shared a ground. To connect all three devices, a quick connect was used. The quick connect allowed the simple connection of pushing the wires into the quick connect, and then squeezing the quick connect closed to join the connections. However, upon testing it was discovered that the ground connects from the quick connects were not connected. The door position sensor and servo was working, but not the RFID. After checking the wiring of the RFID, the conclusion was that the ground was possibly not connected. It was assumed that the wire gauge used for the quick connect was too small, therefore not providing a reliable connection. The solution to this was not to use the quick connects, but to solder all the connections together.

3.3.2 Results

The final look of the cooler as well as system testing is outlined in this section. Systems that needed to be tested include the temperature compartment levels, solar panel power, USB and wireless charging capabilities, car battery charging, and sound level of speakers.

Final Look

The final look of the cooler is as follows. The front of the cooler features the locking mechanism, the Bluetooth speakers, the USB charging ports, light sensor, and the power button.

Fig. 3.30: Smart Cooler Front View

The back of the cooler features simply the solar panel.

Fig. 3.31: Smart Cooler Back View

The side of the cooler features the charging ports for the AC outlet and 12V car port, as well as the flashlight holder and bottle opener.

Fig. 3.32: Smart Cooler Side View

The top of the cooler features the touchscreen and wireless charging pads.

Fig. 3.33: Smart Cooler Top View

The interior features the electronics, LED light strip, and different compartments with lids and temperature sensors.

Fig. 3.34: Smart Cooler Interior View

Power analysis

Looking at the power budget, the system was expected to have a current consumption of about 15A when the entire system is in full operation. However, after testing, realistically the system only consumes about 1A when in full operation. The system current consumption was testing by breaking the circuit at the battery and using a multimeter to measure. The following table show how when a feature is added, the current goes up.

Initially current is about 410 mA with the power on and the Raspberry Pi and touchscreen running.

Table 3.2
System Operation Current Measurements

Component	Total Current (mA)
Startup, Just Raspberry Pi and touchscreen running.	410
Phone Connected to USB Charger	490
One Wireless Charger Used	550
Second Wireless Charger Used	680
Speakers (Max Volume)	770
LED Lights (White)	920

These components affected the batteries the most and as each function was powered, the current consumption increased till it reached about 1A. The system can run with everything powered for about 28 hours. However, if the system is used moderately, it can run for several weeks.

Temperature Monitoring

To test the temperature retention in the cooler, tests were conducted both in an outside and inside environment. First the tests were conducted outside. The tests that needed to be done were to monitor the temperature when there is no ice in the cooler, where there is ice in one compartment only, when there is ice and food in one compartment only, when there is ice in both compartments, when there is ice and food in one compartment and ice in the second compartment, and when there is ice and food in both compartments. The tests with no ice inside the compartments were only run for an hour to provide a control measurement. The rest of the tests were run in a three-hour period and temperatures were recorded every half an hour due to limited time. When ice or food was placed into the compartment, the majority of the space was filled. About 15lbs of ice was placed into each compartment. Fig. 3.35 shows how the compartments with both food and ice were filled.

Fig. 3.35: Both Compartments Filled with Food and Ice

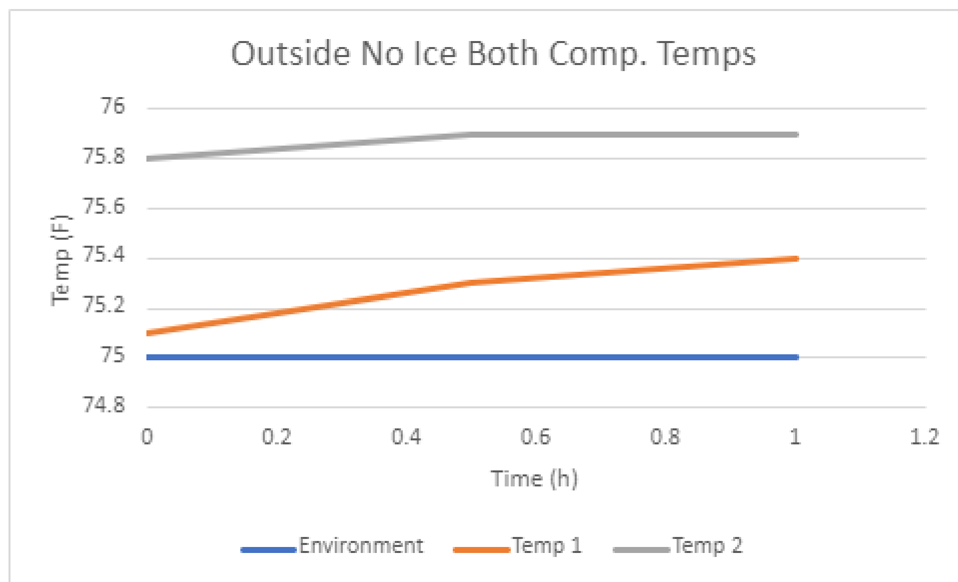


Fig. 3.36: Cooler Outside with No Ice in Compartments Temperatures

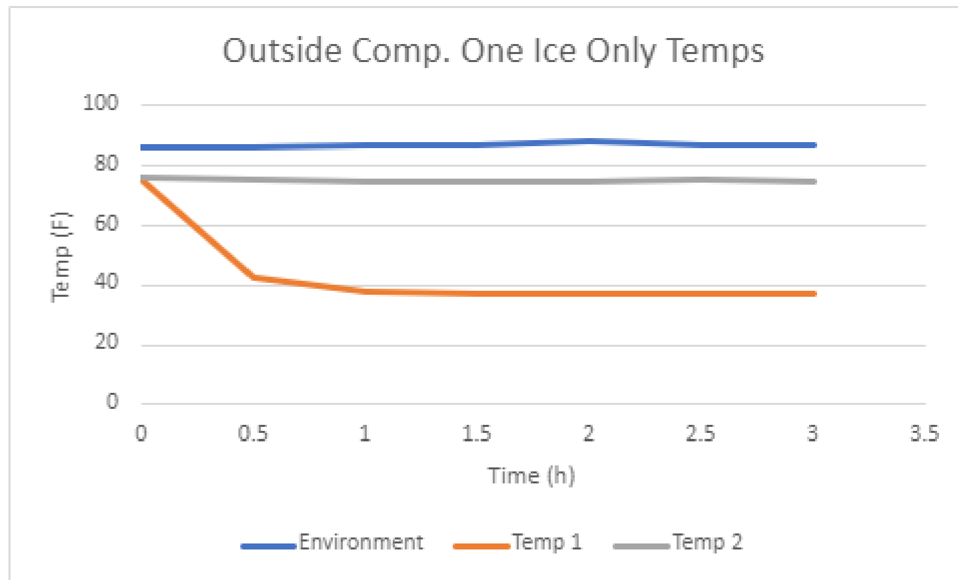


Fig. 3.37: Cooler Outside with Ice in One Compartment Only Temperatures

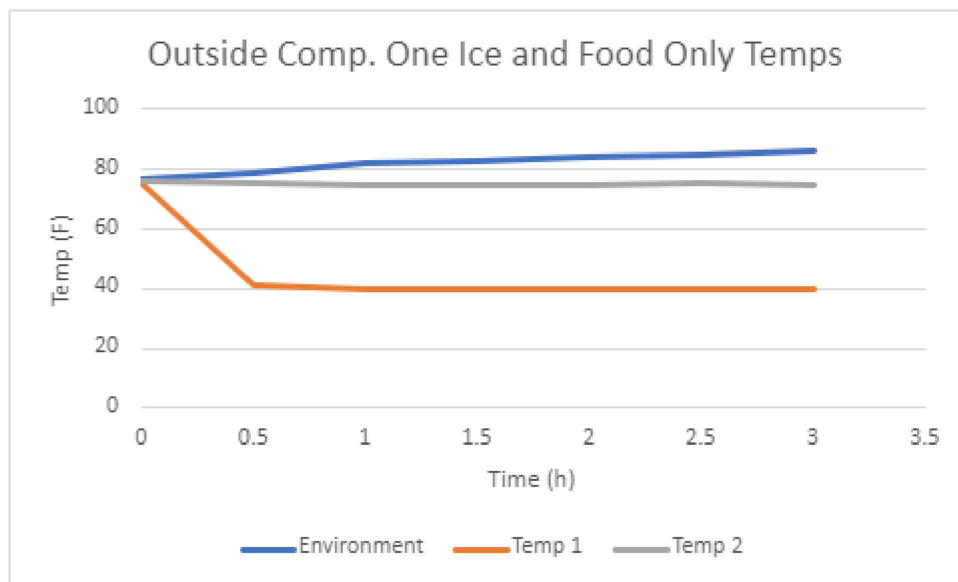


Fig. 3.38: Cooler Outside with Compartment One Ice and Food Only Temperatures

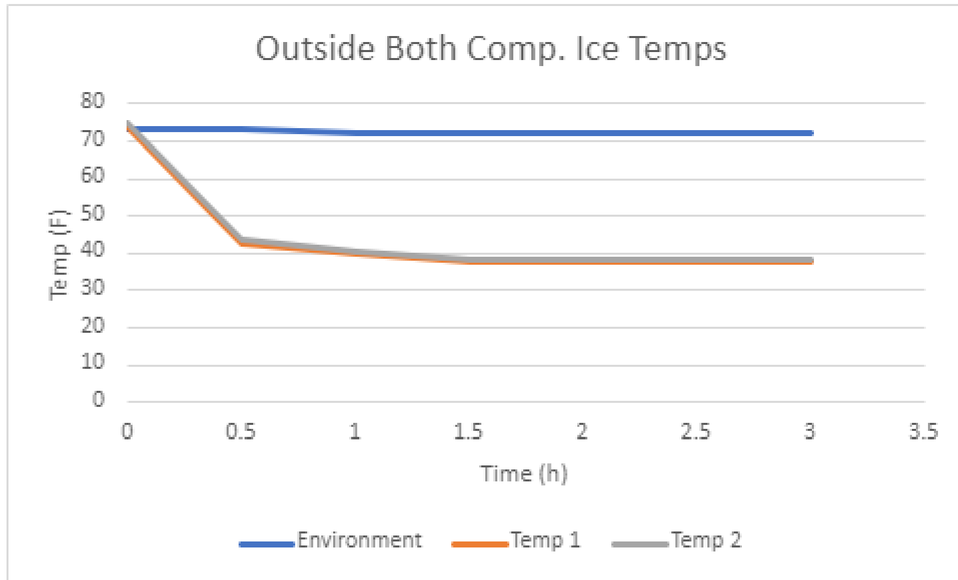


Fig. 3.39: Cooler Outside Both Compartments with Ice Temperatures

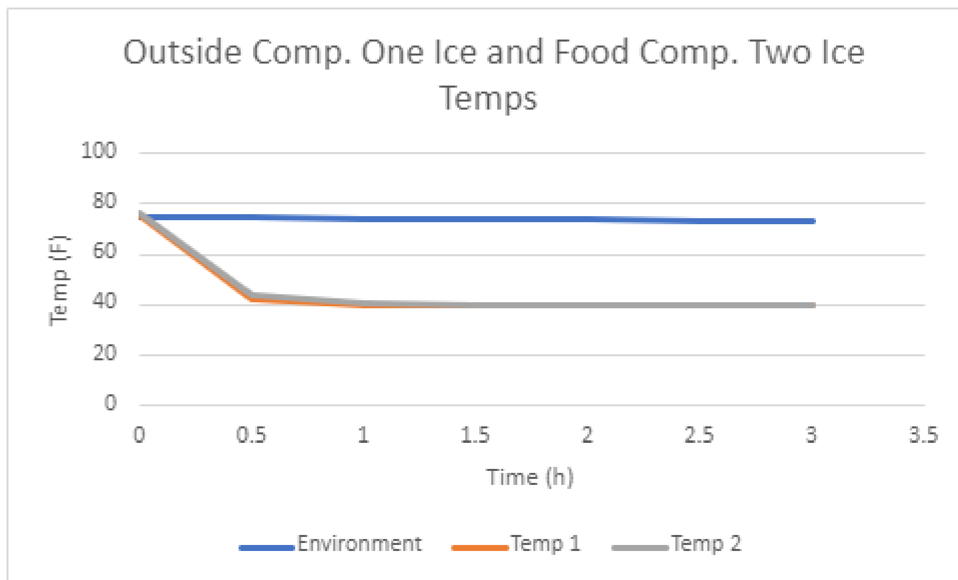


Fig. 3.40: Cooler Outside Compartment One with Ice and Food and Compartment Two with Ice Temperatures

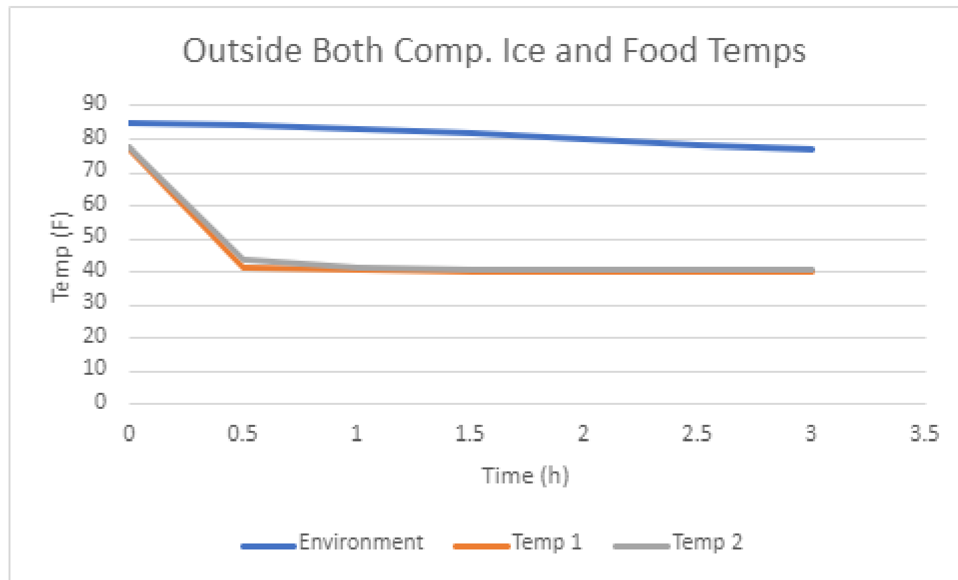


Fig. 3.41: Cooler Outside Both Compartments Ice and Food Temperatures

Note that when both compartments have ice in them, the temperatures are about the same for both compartments. The outside temperature does not affect the inside temperature of the cooler. When ice is in one compartment and no ice in the other compartment, the other compartment does a good job of maintaining its temperature. The lowest temperature reached was 37.6ºF.

Next, the cooler was placed inside a house as the inside environment where the temperature was maintained at 76ºF.

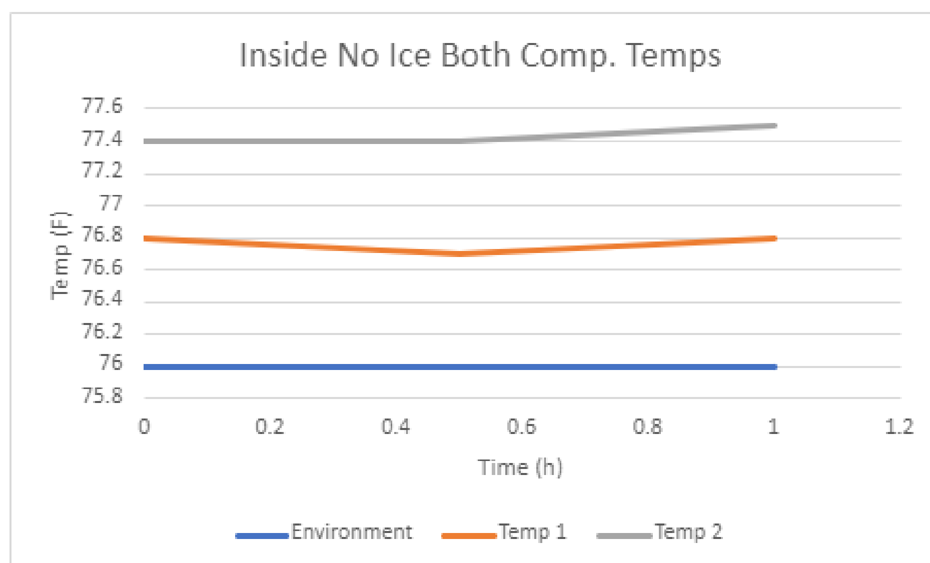


Fig. 3.42: Cooler Inside with No Ice in Compartments Temperatures

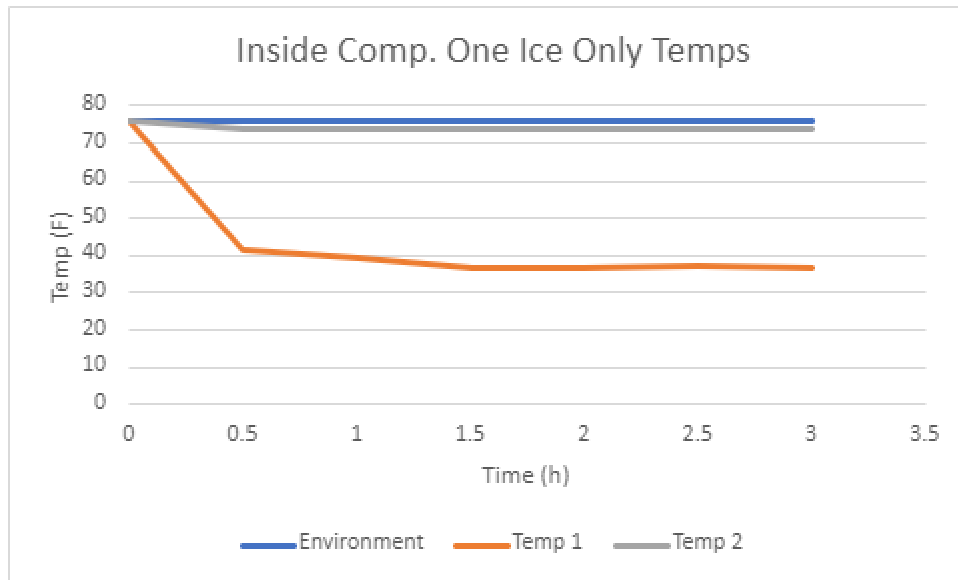


Fig. 3.43: Cooler Inside with Ice in One Compartment Only Temperatures

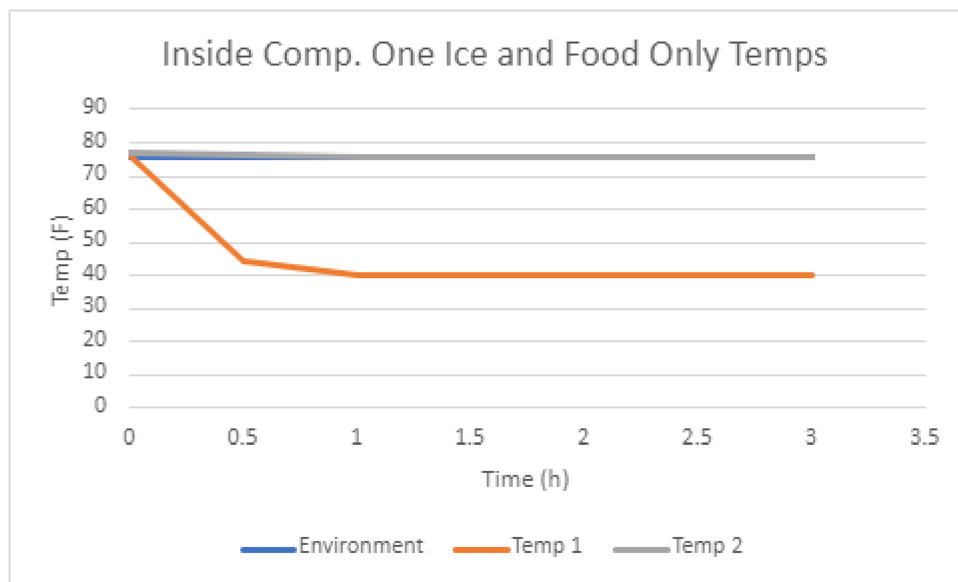


Fig. 3.44: Cooler Inside with Compartment One Ice and Food Only Temperatures

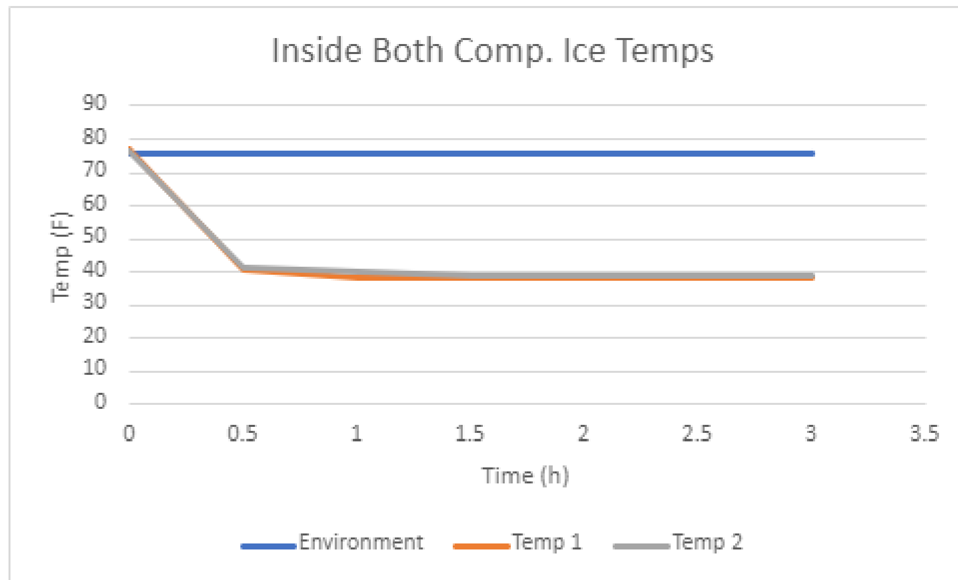


Fig. 3.45: Cooler Inside Both Compartments with Ice Temperatures

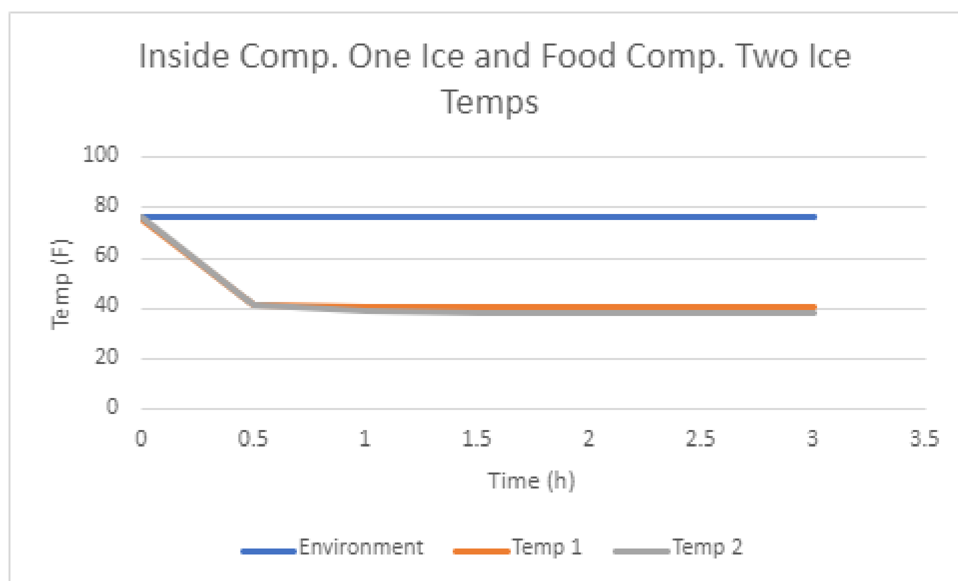


Fig. 3.46: Cooler Inside Compartment One with Ice and Food and Compartment Two with Ice Temperatures

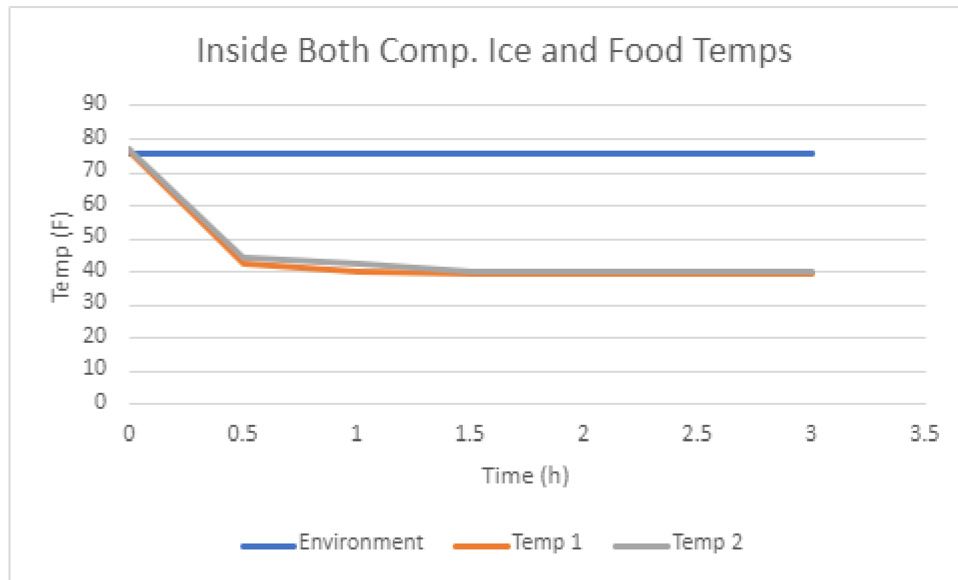


Fig. 3.47: Cooler Inside Both Compartments Ice and Food Temperatures

In all the temperature results for the inside environment were very similar to the outside environment. The lowest temperature reached for the inside environment was 38.4ºF.

Tests were conducted to measure the ice retention and temperature ranges in the cooler for 24 hours. The temperature ranges are in Fahrenheit were measured every three hours for 24 hours. This test featured only ice and there were no drinks or food inside the cooler. The graph in Figure 3.35 shows the temperature maintained in compartment one while the temperature in compartment two was the outside temperature. The ice poured in was only about 7lbs.

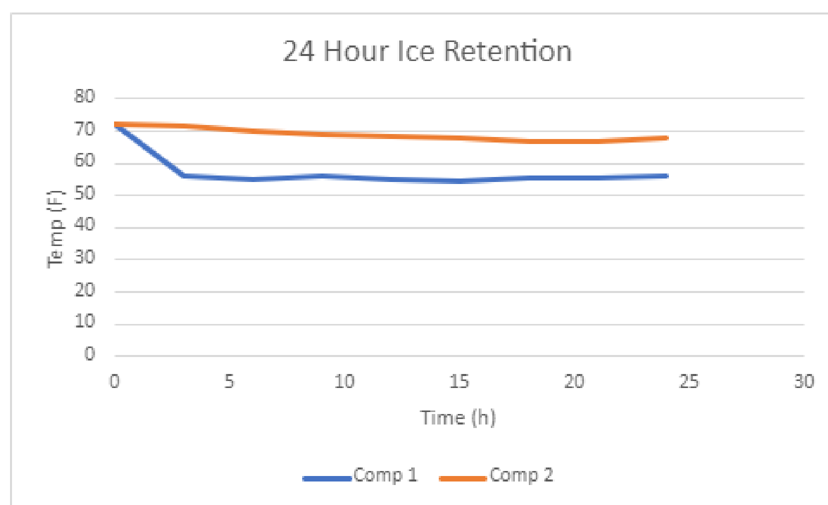


Fig. 3.48: Temperature Measurements in Compartment One (Ice) and Compartment Two (No Ice)

Solar panel charging

The ratings found on the back of the solar panel are shown in the figure below.

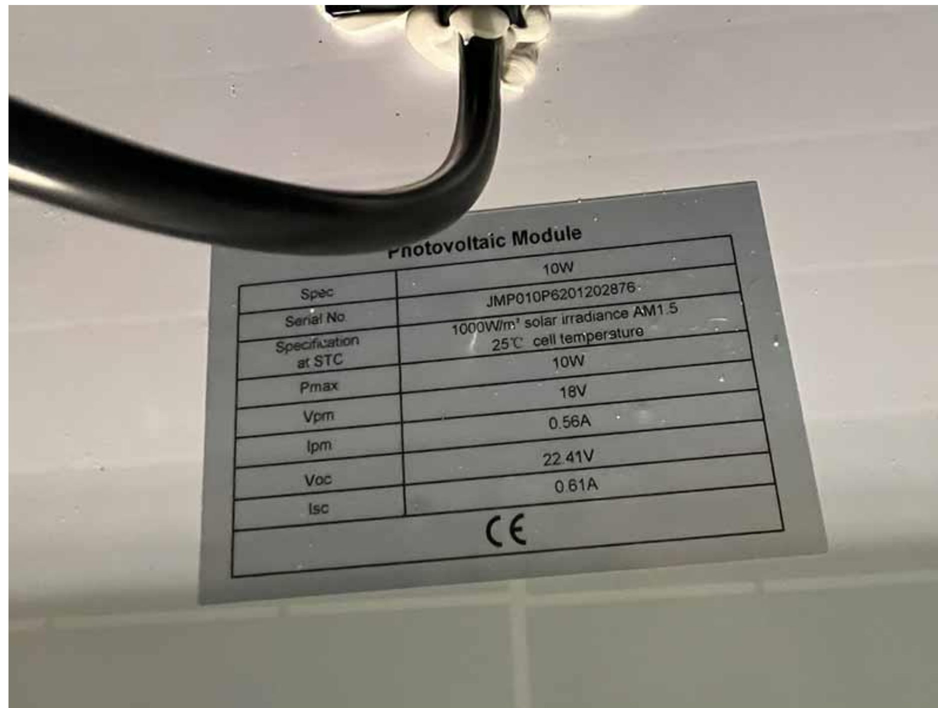


Fig. 3.49: Solar Panel Ratings

The panel is rated to have an open circuit voltage of 22.41V and a short circuit current of 610mA. The panel is also rated for 10W. To measure the open circuit voltage, the solar panel was disconnected, and a multimeter was used to measure the voltage across the panel. The following figure below shows the results, where the voltage was measured during a partly cloudy, which is why there are so many peaks. The max voltage recorded was 21.6V when the sun was exposed but went down the lowest of 18.4V when a cloud covered the sun.

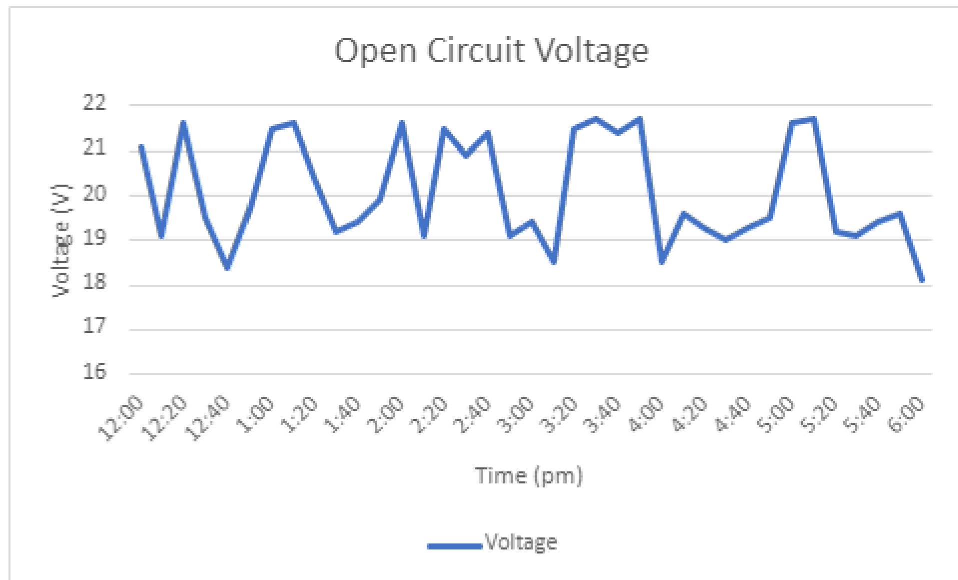


Fig. 3.50: Solar Panel Open Circuit Voltage During Partly Cloudy Day

The short circuit current was measured the same way as the voltage was measured. The highest recorded current measurement was 364mA when the sun was bright and the lowest recorded was 15.2mA when a cloud covered the sun.

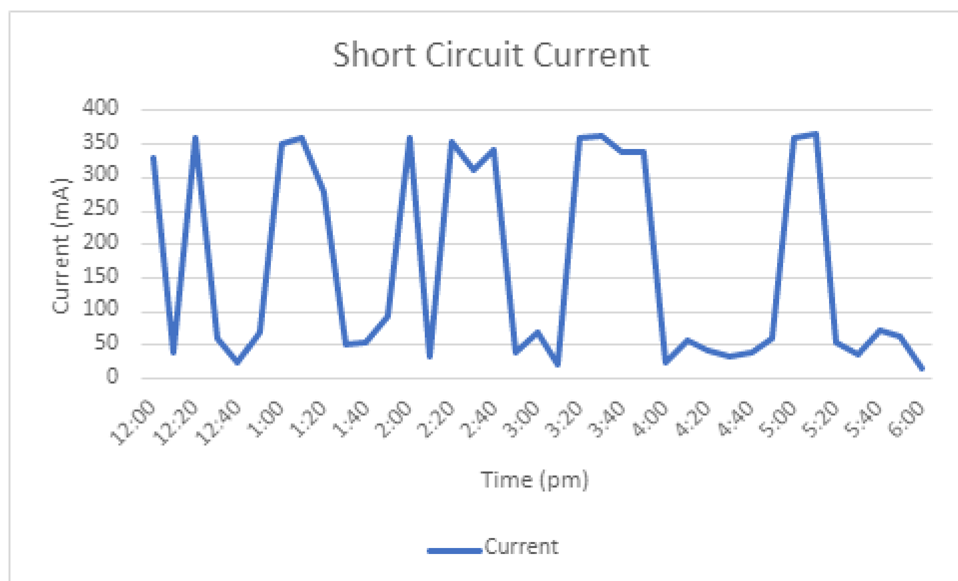


Fig. 3.51: Solar Panel Short Circuit Current During Partly Cloudy Day

The wattage was calculated by using the power equation (2.2) in Appendix A. The max wattage recorded was 7.9W while the lowest recorded wattage was 275mW.

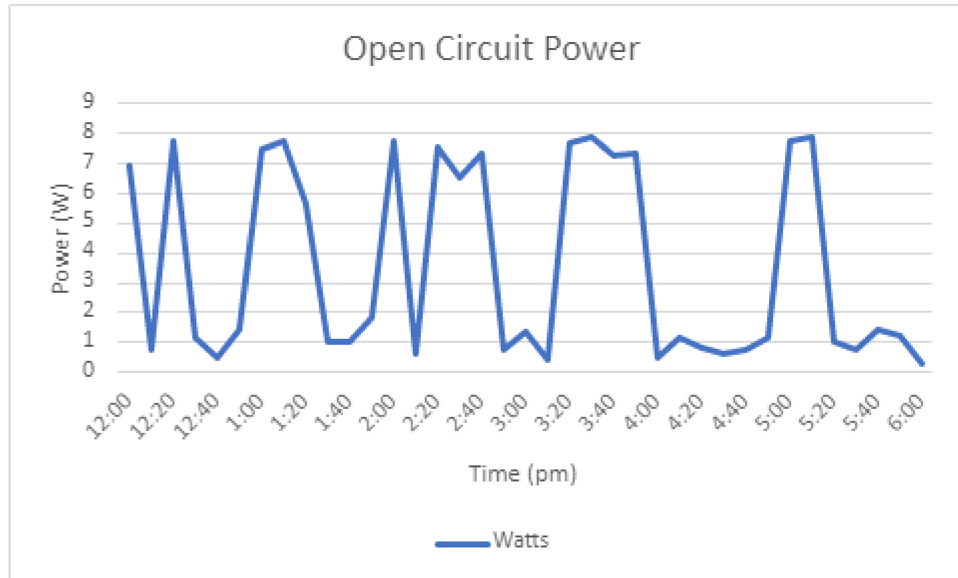


Fig: 3.52: Solar Panel Open Circuit Power During Partly Cloudy Day

Car battery charging

Unfortunately, there were issues trying to get a measurement for the car charging. The car port voltage was measured at 13.6V. However, attempts at measuring the current were not working. The circuit was broken at one of the battery terminals and the multimeter completing the connection, but no current was measured.

Charging of devices

To test the USB charging and wireless charging of the system, an iPhone and Android phone were used. The iPhone was tested on the USB charger on the cooler, while the Android was tested on the wireless charger. The figure below compares using the cooler USB charger, which is 5V 2.1A, and the iPhone power adapter, which is 5.2V 2.4A. The plot shows that the charging capabilities are very similar.

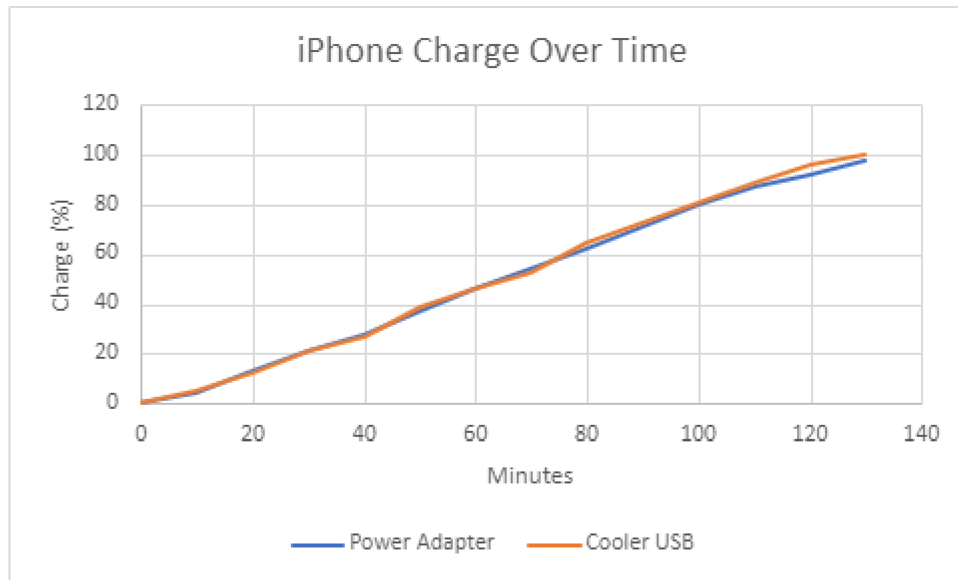


Fig. 3.53: iPhone Charging Percentage Over Time

For the Android, the wireless charging pad was compared to the Android phone power adapter. The wireless charging pad is rated for 5V 1A charging, while the Android power adapter is rated for 5V 2A. It can be seen from the figure below that the power adapter took approximately 100 minutes to charge, while the wireless charging pad took 220 minutes. Since the wireless charging is rated for less amps, it would make sense that the charging takes longer.

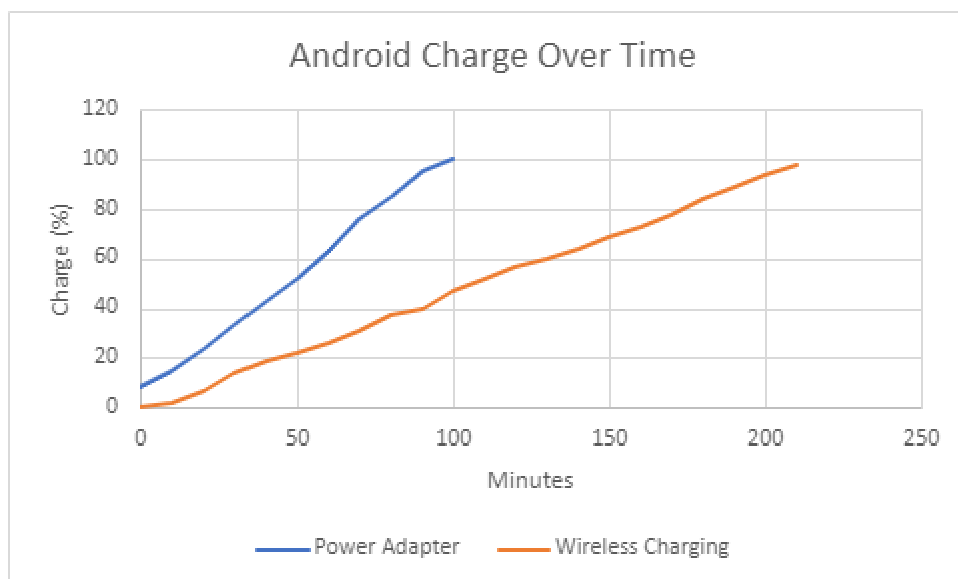


Fig. 3.54: Android Charging Percentage Over Time

Speaker ratings

In order to measure the decibel rating of the Bluetooth speakers, a sound meter app called "Decibel X" was downloaded on an iPhone. The table below shows the decibels for a comfortable listening level at about 78dB. When the speakers were set to max volume, the decibels were at 89dB.

Table 3.3
Speaker Decibel Rating

Distance (feet)	Decibel Rating (dB)
2	78
10	64.8
15	62.3
20	60

3.4 Discussion of Success and Failure Rates

The success criteria are outlined for our project in the following table.

Table 3.4
Success Criteria

Subject	**Success Criteria**
Shell	1) Does the Cooler retain Ice for a minimum of 12 hours? 2) Does the empty weight stay less than 40 lbs? 3) Does the storage volume measure 1.5 cubic feet?
Touch Screen	1) Does the touch screen respond to touch inputs correctly 9/10 times?
Controller	1) Does the microcontroller properly control the system without errors 19/20 times?
App integration	1) Does the App take full control of the system without interference 9/10 times?
Power System	1) Does the system properly power on and off 95% of the time 19/20?

GPS	1) Does the GPS module accurately track the cooler's location 95% of the time?
Locking mechanism	1) Does the lock engage and disengage when signaled by the Touch screen, Mobile app, and RFID keyless entry 9/10 times?

Table 3.5
Final Testing Results

Subject	**Success Criteria**	
Shell	1) Does the Cooler retain Ice for a minimum of 12 hours? 2) Does the empty weight stay less than 40 lbs? 3) Does the storage volume measure 1.5 cubic feet?	21 hours 43 lbs 1.2 cubic ft
Touch Screen	1) Does the touch screen respond to touch inputs correctly 9/10 times?	25/25
Controller	1) Does the microcontroller properly control the system without errors 19/20 times?	20/20
App integration	1) Does the App take full control of the system without interference 9/10 times?	20/20
Power System	1) Does the system properly power on and off 95% of the time 19/20?	50/50
GPS	1) Does the GPS module accurately track the cooler's location 95% of the time?	15/15
Locking mechanism	1) Does the lock engage and disengage when signaled by the Touch screen, Mobile app, and RFID keyless entry 9/10 times?	50/50

The Smart Cooler was able to meet most of the success criteria. The largest and most glaring shortcoming came regarding the storage volume and the weight of the unit. These are mistakes that could have been addressed in the design phase with more emphasis on the size and weight specification of the parts that were researched. The touch screen was tested constantly during the construction of the system and simply while initially was thought to be a point of failure proved to be very consistent. The microcontroller was

accurate did not error or fault during testing. It is worth mentioning that sometimes the controller would slow for a moment but never failed to properly execute the command issued. Once the mobile app was designed it was consistent and error free while connected to the system, which led into the power testing as well. When the Power button was added to the GUI and the power down sequence was set and programmed the unit powered down effectively and did not fault during testing. The GPS module was accurate every time that it was turned on during testing and properly pinned the coolers' location. The locking mechanism was tested using a protocol that was designed to test the RFID reader and the locking mechanism at the same time. The system was locked using the RFID badge, confirmed that it was locked, unlocked with the RFID badge and opened. This protocol was run 25 times meaning that the lock was cycled 50 times using the RFID badge and the other access points were tested during the other operational tests of the system. Overall, the issues that did not meet the success criteria were related to the shell and some of the components selected, not the electronic or mechanical systems.

Chapter 4

Non-Technical issues

Summary

The budget for the project is outlined with the price of each item. The timeline gives the project deadlines for completion of each phase. The project is looked at as a whole as the impact it would have on society, including the effect on the environment, the health and safety impact, ethical and social aspects, and the sustainability of the project.

4.1 Budget
4.2 Timeline
4.3 Environmental Aspects
4.4 Health and safety
4.5 Ethical Aspects
4.6 Social Aspects
4.7 Sustainability

4.1 Budget

One of the requirements for the project was to adhere to a $700 budget, approximately. With the approval of the professor, the budget can go over if requested, but the goal of this project is to stay to the initial budget. However, a lot of modifications and adjustments needed to be made, so the project did run over the budget.

Table 4.1
Actual Project Budget

Modules	Parts	Unit Cost	Quantity	Cost
Structure/Chasis				
Cooler	Coleman 100qt	$69.97	1	$69.97
Micro Controller				
Micro Controller	Raspberry Pi 4	$35.00	1	$35.00
Temperature Regulation				
Temperature Sensor	DS18B20 Waterproof Temp Sensors	$2.20	2	$4.40
Power Control Unit				
Charging Applications	Qi Wireless Charging Transmitter	$26.95	2	$53.90
Solar Panel	ECO-Worthy	$33.00	1	$33.00
Battery	Miadi LFP16AH	$59.99	2	$119.98
Wired USB Charging	YM1236 Dual USB Charger	$9.59	1	$9.59
AC Battery Charger	NOCO Genius10	$64.96	1	$64.96
12V-5V Buck Converter	DROK 5A USB Voltage Regulator	$9.99	1	$9.99
12V-5V Buck Converter	Yipin Hexha DC 5V Converter Module	$14.99	1	$14.99
Locking Mechanism				
RFID	RC 522	$5.49	1	$5.49
Servo	Micro SG90	$5.95	1	$5.95

LED Lights				
LED Light Strip	Alitove WS2812B RGB LED Strip	$23.99	1	$23.99
Photoresistor	Photocell	$0.95	1	$0.95
GPS				
GPS Module	BN-880 GPS Module	$18.99	1	$18.99
Inputs				
TouchScreen	Sunfounder 7 inch	$65.99	1	$65.99
Outputs				
Speakers	Low Profile Marine Speakers	$29.99	1	$29.99
Amp board	TPA3116,DAMGOO	$22.99	1	$22.99
Other Components				
Analog to Digital Converter	MCP3008	$3.75	1	$3.75
Misc				
MOSFET	FQP30N06 N-Channel MOSFET	$0.90	3	$2.70
Lid Position Sensor	Magnetic Reed Switch	$5.00	1	$5.00
Terminal Block	600V 25A Dual Row Screw Terminal	$2.83	2	$5.66
Insulation Shield	3MM Reflective Foam Insulation	$16.88	1	$16.88
Connector Plug	LanHong 2 Pin Connector Plug	$1.20	2	$2.40
Foam Board	1" Thick Polystyrene Foam Board	$20.99	1	$20.99
USB Cable	DTECH Type A to A USB Cable	$6.98	2	$13.96
Power Switch	Magic&Shell Round Rocker Switch	$1.75	1	$1.75
Drain Plug	Cooler Drain Plug Replacement	$11.99	1	$11.99
Plexiglass	Optix Plexiglass	$34.00	2	$68.00
3d Printing Filament	1.75mm PLA Filament 1kg	$20.00	1	$20.00
Solar Panel Metal Bracket	Armstrong 2' Cross Tee	$1.90	1	$1.90
Sliding Lock	ReliaBilt Zinc Steel Barrel Bolt	$2.68	1	$2.68

Wooden Dowel	Wooden Dowel	$1.50	1	$1.50
Various Screws and Bolts	Various Screws and Bolts	$5.00	1	$5.00
Total				$774.28

4.2 Timeline

The timeline gives start dates and end dates for each phase of the project. The original project timeline was extended as the project was not complete for the first semester taken for Senior Design Project. Table 4.3 reflects the extended timeline.

Table 4.2
Senior Design Proposal Timeline

Task	Length (Days)	Start Date	End Date
Research Ideas	17	05/12	05/28
Research Components	51	05/18	07/09
Proposal Report	26	06/07	07/02
Proposal Presentation	7	07/02	07/09
Proposal Website	59	05/12	07/09

Table 4.3
Senior Design Project Timeline

Task	Length (Days)	Start Date	End Date
Buy Parts	19	07/12/2021	07/31/2021
Part Testing and Validation	13	08/01/2021	08/14/2021
Individual Subsystem Testing	44	08/07/2021	09/20/2021
Fabricate and Installation	175	09/01/2021	02/23/2022
Mobile App Development	221	08/01/2021	03/10/2022

OS Software Development	221	08/01/2021	03/10/2022
Final Unit Testing	26	03/20/2022	04/15/2022
Project Report	64	02/10/2022	04/15/2022
Project Presentation	21	04/01/2022	04/22/2022
Project Website	264	08/01/2021	04/22/2022

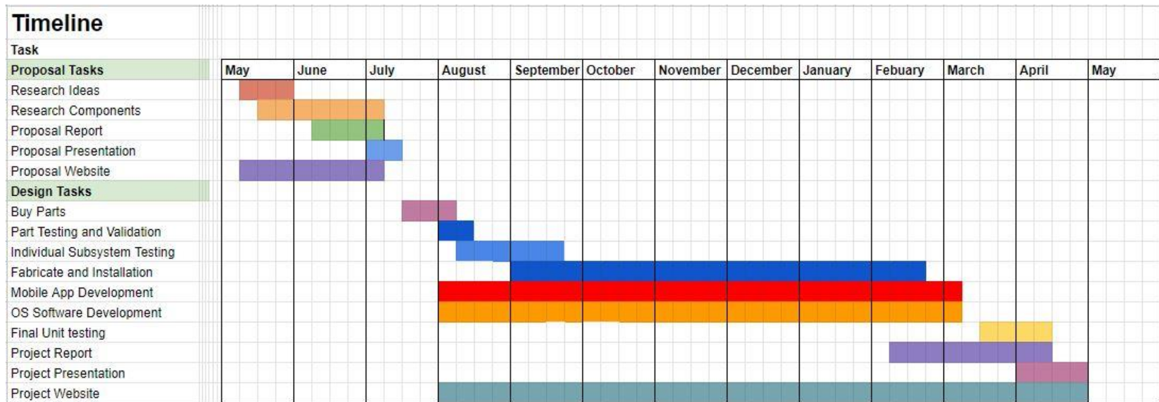


Figure 4.1: Graphical Representation of Timeline

4.3 Environmental Aspects

In order to keep the system environmentally friendly and for convenience of the user, the cooler has 12V rechargeable LiFePO4 batteries. Lithium batteries have more cycles than SLA batteries, which means it will last longer. Having rechargeable batteries reduces the waste of disposable batteries and has renewable energy benefits. Since the Smart Cooler has a 12V solar panel attached to it, the batteries will be able to be recharged from the sun. This can also be extended to the AC input as well. If the user’s house has solar panels, then the cooler can be recharged through the limitless power of the sun, leaving no carbon footprint.

4.4 Health and Safety

The health and safety of the user is the most important aspect for this project. That is why extensive research was conducted into the power source. Making sure the batteries are worked within their safety specifications, as well as proper charging and discharging, making the cooler waterproof so components don’t get damaged, proper ventilation to

maintain a safe operating temperature for all the components, and extensive testing of components individually and assembled ensured a safe a reliable product that can be distributed without worry.

4.5 Ethical Aspects

Holding to the IEEE Code of Ethics, the "safety, health, and welfare of the public" is the top priority [30]. The Smart Cooler attempts to use technology to improve the functionality of an item that has been used for decades with minimal improvements and advancements. Given that the health and safety of the user is the top priority, testing of the product along with the success criteria specified will ensure that everything is working properly, and should the system fail, it will not do so in a way that endangers the user or surroundings.

All references and materials used in this project have been cited and proper credit given to the original sources.

4.6 Social Aspects

Party engagements are the ideal scenario for the Smart Cooler. Having an all-purpose party cooler could possibly trend because of its utility, and possibly allowing something like a smart cooler to become ubiquitous.

Having GPS to find the location of the cooler will prevent people from getting lost in the woods. "Around 2,000 people get lost in the woods every year" [31]. Not everyone in the woods knows wilderness survival skills necessary to find their campsite.

4.7 Sustainability

The batteries used are 2000 deep cycle LiFePO4 batteries. Lithium batteries have the advantage of having more cycles, faster charge time, and are lightweight compared to SLA batteries. Lithium batteries last longer, so therefore the sustainability of the product was taken into consideration. If the batteries were to be charged every day, they would last 5 and a half years.

Chapter 5

Conclusion

Summary

The long list of features that the Smart Cooler has went through a lot of research and analysis. With this list of wanted features, components were researched, specifications were developed, and success criteria were specified. The cooler has a lot of possible implementations and can only be improved upon in the near future.

5.1 Summary and Conclusion

5.2 Suggestions for Future Work

5.1 Summary and Conclusion

The Smart Cooler is bound to be a successful product. The success of the INFINITE cooler and The Coolest Cooler shows that demand for a multipurpose cooler is plentiful. Each cooler product had its own set of features that allowed convenience for all sorts of possibilities. There are many different situations that a cooler can be used for, so to cater for all possibilities is the goal.

Given that the Smart Cooler will have a battery source for power, this allows the addition of many features. The smart capabilities added to this cooler makes it a more modern solution to your typical gathering. The power budget shows how resources will be allocated, and the batteries selected provide plenty of power that allows the cooler to operate for days. This functionality was built and tested for safety. All the modules have certain criteria that have been met, and testing ensured that every module is working as intended. On initial iteration of any product, the development cost is usually higher than when something is mass produced. So even though this design went over the initial budget, future iterations could drop the cost and make it more affordable, as was done with the INFINITE cooler and the Coolest Cooler.

Having a cooler with smart capabilities has many positive implications in society. It can be the object of the party, the storage to preserve food, or even used in other scenarios besides just parties. Whatever the use case, the aim of the project is to improve upon the typical cooler, giving more functionality that turns something simple to an advanced, smart item.

5.2 Suggestions for Future Work

The main suggested improvement for this type of project is to prolong the temperature regulation. By adding more batteries, or batteries with more Ah ratings, thermoelectric refrigeration units can be added to the cooler, and allowing temperature regulation and ice retention for longer than seven days would be a tremendous improvement. However, adding more batteries, and more powerful batteries at that, will increase the price of the product. Of course, additional features can be included as there will always be something that can be added. One important feature that can be added is for the cooler to have a way to contact emergency services. For the cooler to have that feature, it would need a SIM card, unfortunately.

References

Electronic Sources from Internet

[1] "INFINITE: World's Most Versatile, Smart Cooler," Indiegogo. [Online] Available at: https://www.indiegogo.com/projects/infinite-world-s-most-versatile-Smart-cooler#/ (Accessed June 17, 2021)

[2] R. Grepper. "Coolest Cooler: 21st Century Cooler That's Actually Cooler," Kickstarter, 03/12/2018. [Online] Available at: https://www.kickstarter.com/projects/ryangrepper/coolest-cooler-21st-century-cooler-thats-actually (Accessed June 17, 2021)

[3] K. Schlosser. "Coolest Cooler shuts down after 5-year saga, leaving 20,000 backers without Kickstarter reward," GeekWire, 12/09/2019. [Online] Available at: https://www.geekwire.com/2019/coolest-cooler-shuts-5-year-saga-leaving-20000-backers-without-kickstarter-reward/ (Accessed June 17, 2021)

[4] D. Kliszowki, A. Vlasov. "Bluetooth Classic vs. Bluetooth Low Energy (BLE) on Android – Hints and Implementation Steps," Droid on Roids, 04/07/2020, [Online] Available at: https://www.thedroidsonroids.com/blog/bluetooth-classic-vs-bluetooth-low-energy-ble (Accessed March 30, 2022)

[5] "Bluetooth Wireless Technology," Bluetooth. [Online] Available at: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/ (Accessed March 30, 2022)

[6] M. Woolley. "A Developer's Guide to Bluetooth Technology," Bluetooth, 08/10/2016, [Online] Available at: https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/ (Accessed March 30, 2022)

[7] "Coleman 100 Quart Xtreme 5 Wheeled Cooler," Amazon. [Online] Available at: https://www.amazon.com/Coleman-100-Quart-Xtreme-Heavy-Duty-Cooler/dp/B000G64FJK/ (Accessed July 5, 2021)

[8] "Raspberry Pi 4 Model B 2019 Quad Core 64-bit WiFi Bluetooth," Amazon. [Online] Available at: https://www.amazon.com/Raspberry-Model-2019-Quad-Bluetooth/dp/B07TC2BK1X (Accessed July 5, 2021)

[9] "Pyle Marine Waterproof Speakers 6.5" - Low Profile Slim Style Wakeboard Tower and Weather Resistant Outdoor Audio Stereo Sound System with LED Lights and 240 Watt Power - 1 Pair in Black - PLMRS63BL," Amazon. [Online] Available at: https://www.amazon.com/Pyle-Marine-Waterproof-Speakers-6-5/dp/B078JBSPRJ (Accessed July 5, 2021)

[10] "Eiechip 5PCS Waterproof Temperature Probe Thermometer DS18B20 1M with Heat Resistance Thermal Cable," Amazon. [Online] Available at: https://www.amazon.com/gp/product/B07MB1J43W (Accessed March 30, 2022)

[11] “MCIGICM 30 Pcs Photoresistor Photo Light Sensitive Resistor, Light Dependent Resistor 5 mm GM5539 5539,” Amazon. [Online] Available at: https://www.amazon.com/MCIGICM-Photoresistor-Sensitive-Resistor-Dependent/dp/B07PF3CWW9 (Accessed July 5, 2021)

[12] “Upenten 1pc SG90 9g Micro Servo Motor for RC Robot, Helicopter, Drone, Car, Boat Remote Control,” Amazon. [Online] Available at: https://www.amazon.com/Upenten-Micro-Helicopter-Remote-Control/dp/B07KVJ84FS/ (Accessed March 30, 2022)

[13] “Servo Motor SG-90,” Components101, 09/18/2017. [Online] Available at: https://components101.com/motors/servo-motor-basics-pinout-datasheet (Accessed March 30, 2022)

[14] MIT Electric Vehicle Team. “A Guide to Understanding Battery Specifications,” MIT, 12/2008. [Online] Available at: http://web.mit.edu/evt/summary_battery_specifications.pdf (Accessed June 21, 2021)

[15] “NOCO Genius10, 10-Amp Fully-Automatic Smart Charger, 6V and 12V Battery Charger, Battery Maintainer, Trickle Charger, and Battery Desulfator with Temperature Compensation,” Amazon. [Online] Available at: https://www.amazon.com/NOCO-GENIUS10-Fully-Automatic-Temperature-Compensation/dp/B07W3QT226 (Accessed July 5, 2021)

[16] “ECO-WORTHY 10W 12V Off Grid Small Solar Panel Kit - Waterproof 10 Watt Solar Panel with Charge Controller and Battery Clips Adapter,” Amazon. [Online] Available at: https://www.amazon.com/ECO-WORTHY-Polycrystalline-System-Controller-Battery/dp/B00PFG56ZS (Accessed July 5, 2021)

[17] “12V 20Ah Deep Cycle LiFePO4 Battery, 2000 Cycles Miady LFP16AH Rechargeable Battery, Maintenance-Free Battery for Golf Cart, Boat, Solar System, UPS and More,” Amazon. [Online] Available at: https://www.amazon.com/LiFePO4-Battery-Miady-Rechargeable-Maintenance-Free/dp/B089VXSBC6/ (Accessed July 5, 2021)

[18] “BN-880 GPS Module U8 with Flash HMC5883 Compass + GPS Active Antenna Support GPS Glonass Beidou Car Navigation for Arduino Raspberry Pi Aircraft Pixhawk APM Flight Controller Geekstory,” Amazon. [Online] Available at: https://www.amazon.com/Geekstory-Navigation-Raspberry-Aircraft-Controller/dp/B078Y6323W/ (Accessed July 5, 2021)

[19] “DAMAVO YM1218 USB C & USB A Dual Port Car Charger Socket Power Outlet Adapter Type C USB Car Charger Socket Waterproof with Cap 12V/24V for Car, Boat, Golf Cart, Bus, RV, Automotive Marine ATV Truck,” Amazon. [Online] Available at: https://www.amazon.com/dp/B08DD4BH6G/?coliid=ID6SK5UNY8K9F&colid=2J10L2RGHQ1NS&psc=1&ref_=lv_ov_lig_dp_it (Accessed July 5, 2021)

[20] “ALITOVE 16.4ft WS2812B Addressable Programmable Digital RGB LED Strip Light 30 LED/M 5m 150 Pixels Dream Color LED Flexible Strip Waterproof IP65 5VDC for Home Bedroom Bar Décor Lighting,” Amazon. [Online] Available at: https://www.amazon.com/dp/B07FVPN3PH (Accessed March 30, 2022)

[21] "RC522 RFID Module," Components101, 06/12/2019. [Online] Available at: https://components101.com/wireless/rc522-rfid-module (Accessed March 30, 2022)

[22] "HiLetgo 3pcs RFID Kit – Mifare RC522 RF IC Card Sensor Module + S50 Blank Card + Key Ring for Arduino Raspberry Pi," Amazon. [Online] Available at: https://www.amazon.com/dp/B07VLDSYRW (Accessed March 30, 2022)

[23] "DC Converter with Battery Clip 12V 24V to 5V 8A USB Power Adapter Buck Regulator Charger, 4 Ports Automatically Identify Shunt Charging, Suitable for iPhone Android Samsung Galaxy S10 S9 Plus," Amazon. [Online] Available at: https://www.amazon.com/gp/product/B08MCZNYZN (Accessed March 30, 2022)

[24] "Buck Converter 12V to 5V, DROK 5A USB Voltage Regulator DC 9-36V Step Down to DC 5V-5.3V 5.2V 3.5-6A Volt Transformer Power Supply Module for Phone Fast Charging," Amazon. [Online] Available at: https://www.amazon.com/gp/product/B01NALDSJ0 (Accessed March 30, 2022)

[25] "BOJACK FQP30N06 32A 60V MOSFET Transistors FQP30N06L 32 Amp 60 Volt N-Channel Power MOSFET TO-220AB (Pack of 10 Pcs)," Amazon. [Online] Available at: https://www.amazon.com/gp/product/B08B8WRQP1 (Accessed March 30, 2022)

[26] "Universal Qi Wireless Charging Transmitter," Adafruit. [Online] Available at: https://www.adafruit.com/product/2162?gclid=Cj0KCQjw24qHBhCnARIsAPbdtlIAUh2ibQHBZja9W_nasyJGfjjIFm_Lz3AFodz734EUntLDn1SxIU4aApN7EALw_wcB (Accessed July 5, 2021)

[27] "SunFounder Raspberry Pi 4 Display Touchscreen 7 Inch HDMI 1024x600 USB IPS LCD Screen Display Monitor for Raspberry Pi 400 4 3 Model B, 2 Model B, and 1 Model B+, Windows Capacitive Touch Screen," Amazon. [Online] Available at: https://www.amazon.com/SunFounder-Raspberry-Touchscreen-1024%C3%97600-Capacitive/dp/B07Y889J3X (Accessed July 5, 2021)

[28] "2Sets Magnetic Reed Switch Normally Open Closed NC NO Door Alarm Window Security/Magnetic Door Switch/Magnetic Contact Switch/Reed Switch for GPS, Alarm or Other Device, DC 5V 12V 24V Light," Amazon. [Online] Available at: https://www.amazon.com/gp/product/B0735BP1K4 (Accessed March 30, 2022)

[29] J. Anderson. "An Intro to Threading in Python," Real Python. [Online] Available at: https://realpython.com/intro-to-python-threading/ (Accessed March 30, 2022)

[30] "IEEE Code of Ethics," IEEE. [Online] Available at: https://www.ieee.org/about/corporate/governance/p7-8.html (Accessed July 5, 2021)

[31] B. Conghalie. "What To Do If You Get Lost In The Woods," MyOpenCountry, 03/17/2021. [Online] Available at: https://www.myopencountry.com/lost-woods/ (Accessed at June 25, 2021)

Appendix A

Equations

Celsius to Fahrenheit Equation $$°F = °C \cdot \frac{9}{5} + 32 \tag{2.1}$$

Power Equation $$P = V \cdot I \tag{2.2}$$

Appendix B

Smart Cooler Survey

Summary

This survey was posted online to get feedback on the proposed features, as well as additional features that would like to be added and where the Smart Cooler would be used. The survey was kept short to encourage more responses.

Question 1: What features would you be most excited about?

Smart Cooler Features

For my senior design project I am building a smart cooler. This survey will help to find what features would be most popular or additions that can be added.

What feature would you be most excited to have?

- Temperature Regulation
- Bluetooth Speakers
- Wireless Charging
- Monitor
- GPS Tracking
- Other

Question 2: What features would you like in a smart cooler?

What features would you like in a smart cooler?

Your answer

Question 3: Where would you use the cooler most often?

Where would you use the cooler most often?

- ◯ Beach
- ◯ Camping
- ◯ Work
- ◯ Other: ______

Appendix C

Smart Cooler Survey Results

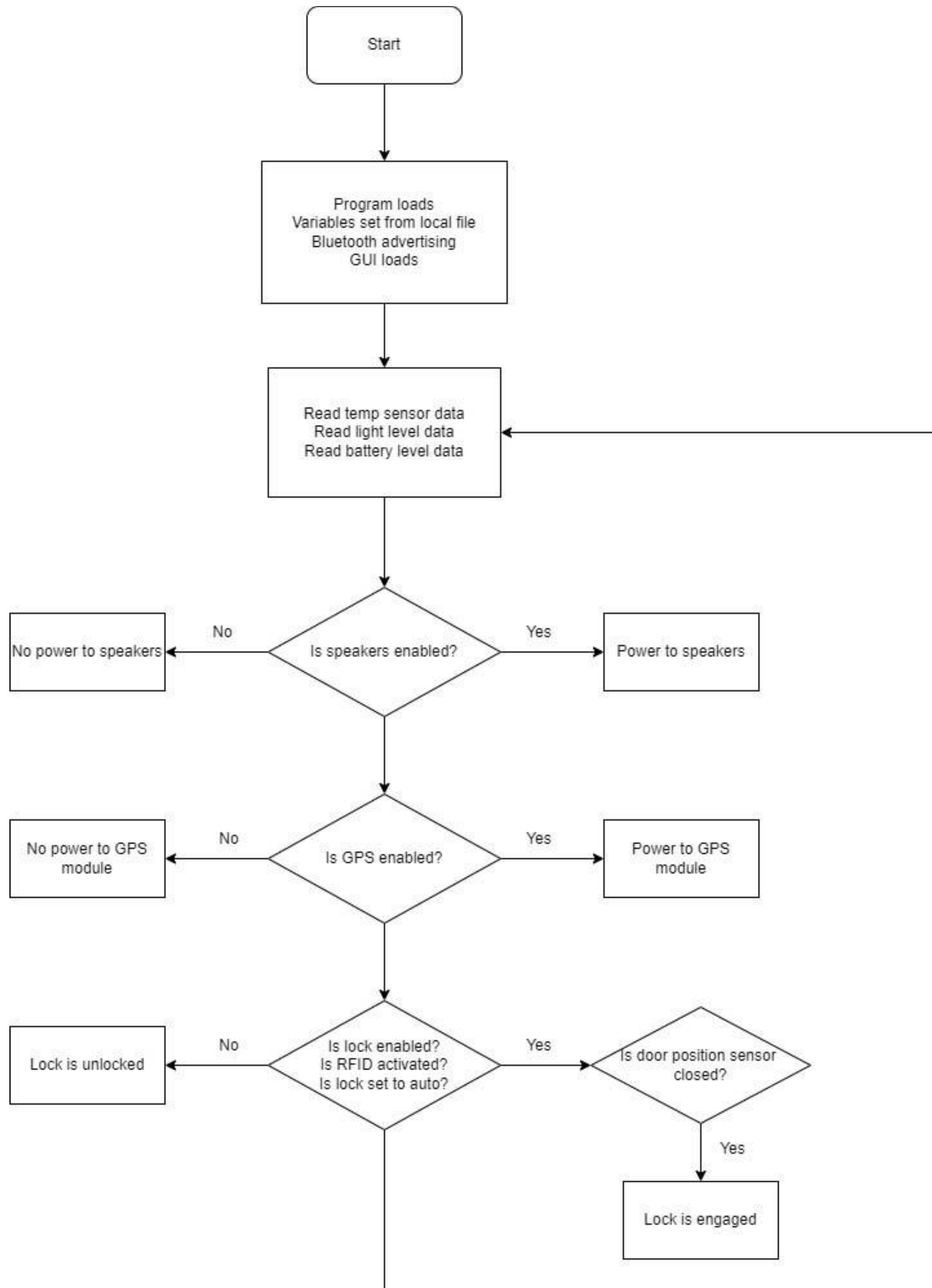
Timestamp	What feature would you be most excited to have?	What features would you like in a smart cooler?	Where would you use the cooler most often?
6/5/2021 15:22:25	Bluetooth Speakers	The ability to stay cold for a really long time	Beach
6/5/2021 15:45:24	Temperature Regulation	Separate small freezer compartment(in conjunction with temperature regulation)	Beach
6/5/2021 15:45:49	Bluetooth Speakers	App support	Beach
6/5/2021 16:04:03	Temperature Regulation		what is a cooler?
6/5/2021 16:12:27	Temperature Regulation	The most important features for me when buying a smart cooler would be a long-lasting battery and a way to regulate/monitor the temperature. It would also need other basic features like being waterproof and having adequate storage room.	Camping
6/5/2021 16:42:19	Wireless Charging		Camping
6/5/2021 17:02:02	Temperature Regulation	Lightweight, easily mobilised, good temperature regulation, reasonably priced, low maintenance	Camping
6/5/2021 17:11:24	Temperature Regulation	Probably most useful would be ability to control temperature inside cooler from an app and check what's inside the cooler	Beach
6/5/2021 17:34:01	Temperature Regulation	Cooling	
6/5/2021 17:43:40	Other	None at all	Beach
6/5/2021 17:50:42	Bluetooth Speakers		Camping
6/5/2021 17:54:37	Temperature Regulation		Camping
6/5/2021 18:04:12	Temperature Regulation	Charging would also be nice	Beach
6/5/2021 18:10:22	Temperature Regulation	wheels that work well on multiple types of surfaces, for example asphalt and sand	picnic
6/5/2021 18:30:02	Wireless Charging	Durable	Beach
6/5/2021 18:39:00	Temperature Regulation	less bulky, when i have to pack up my cooler and take it out for work it's a giant pain something slimmer that fits better in a laptop bag	Work
6/5/2021 18:39:46	Temperature Regulation	Price	Camping

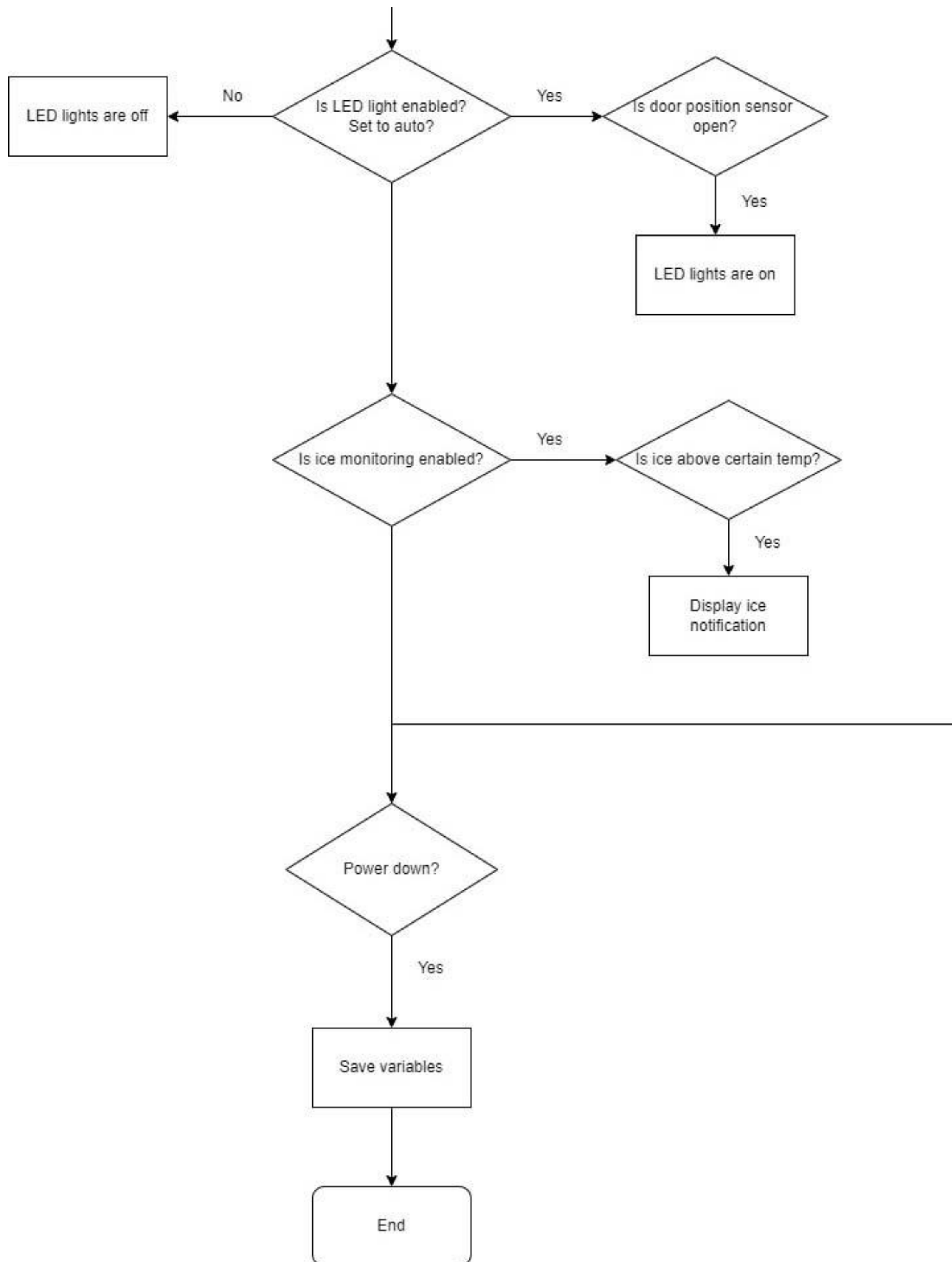
6/5/2021 19:13:25	Temperature Regulation	Temp monitoring through app. Alarm if it drops under a programmable temp. Removable freezer blocks that can create dividers, shelves, and can intigrate into the lid to have cold come down from top.	Camping
6/5/2021 19:16:47	Temperature Regulation		Beach
6/5/2021 19:19:12	Temperature Regulation	maintain different temperatures for different items	Work
6/5/2021 19:46:37	Temperature Regulation	On screen thermometer.	Work
6/5/2021 20:24:01	Wireless Charging		Camping
6/5/2021 20:37:51	Temperature Regulation	I want it to be as waterproof and sandproof as possible, sturdy enough to sit on, spacious enough for all the beers I'm going to drink, bluetooth speakers would be nice, solar chargeable with attachable panel, plug in capability, but it's not necessary to have it plugged in all the time to keep things cool, lightweight for carrying, made of recycled materials if possible, heck, maybe even a cute little screen that you can play games on to occupy the kids in the back seat.	Camping
6/5/2021 20:39:52	Temperature Regulation		Camping
6/5/2021 20:44:57		keep my drinks cold. don't leak	house party
6/5/2021 20:46:43	Temperature Regulation		car
6/5/2021 21:05:22	Temperature Regulation		Beach
6/5/2021 21:11:33	Temperature Regulation	Intelligent temperature control; swappable inserts for different types of items like holding drinks upright or shelves for more varied items	Work
6/5/2021 21:20:42	Monitor		Camping
6/5/2021 21:31:42	Temperature Regulation		Beach
6/5/2021 23:47:06	Temperature Regulation	Temperature sensor, readout that gives a warning and timestamp of when the cooler has exceeded the allowable internal temp (so you know when to buy more ice or if foods are safe to eat while camping)	Grocery shopping to transport frozen goods
6/6/2021 1:41:29	Temperature Regulation	Thermometer reading of the inside temperature. Perhaps even a way to set the cooler to a specific temperature. GPS tracking and a way to contact emergency services. In the event that something goes wrong while camping and for some reason the people camping do not have access to their phones or any other emergency device. Perhaps make it animal-proof so that animals can't easily get inside and maybe a way to repel wild animals.	Outdoor work and activities
6/6/2021 1:52:29	Temperature Regulation		Travel

6/6/2021 2:44:40	Wireless Charging	Notification when ice is melted	On a boat
6/6/2021 3:38:19	Temperature Regulation		Beach
6/6/2021 7:35:28	Temperature Regulation	Digital temp controls and easy portability	Road trips and days at the park
6/6/2021 14:46:04	Temperature Regulation		Camping
6/6/2021 19:04:56	Temperature Regulation	Temperature regulation, wheels, cup holder	Beach
6/6/2021 22:12:36	Temperature Regulation	An app or some thing that lets you control the temperature. Also cool LED lights inside of it	Beach
6/6/2021 22:26:29	Bluetooth Speakers	Bluetooth speaker & wireless charging	Everywhere: home, picnic, etc.
6/6/2021 22:31:04	Temperature Regulation		Beach
6/6/2021 23:44:57	Temperature Regulation	Wireless charging and Bluetooth speakers	Family gatherings
6/7/2021 9:27:46	Temperature Regulation	Monitor for Perishable items	Camping
6/8/2021 0:17:12	Bluetooth Speakers		Beach
6/8/2021 15:54:14	Bluetooth Speakers	Mobility, BT, water proof	Beach
6/8/2021 17:04:15	Bluetooth Speakers	Flashlight	Camping
6/9/2021 17:02:10	Other	Must cost less and use less power then non-smart cooler. Must work without additional app/smartphone/internet	
6/30/2021 8:31:26	Temperature Regulation	Keeping track of the optimum temperature for the drinks. Like if we put stuff in it, if it can tell when drinks are cooled down to its most chill temperature to enjoy.	Beach

Appendix D

Flowchart




No
Is LED light enabled?
Set to auto?
Yes
LED lights are off
Is door position sensor open?
Yes
LED lights are on
Is ice monitoring enabled?
Yes
Is ice above certain temp?
Yes
Display ice notification
Power down?
Yes
Save variables
End

Appendix E

Raspberry Pi Software Code

Reuben_app_LED.py is main file program.

```
#!/usr/bin/env python3

###############################################
# GPS
# First, to get GPS working, open terminal and run
# sudo systemctl stop gpsd.socket
# sudo gpsd /dev/serial0 -F /var/run/gpsd.sock

###############################################
# TO RUN LEDS, RUN PYTHON SCRIPT IN 'SUDO PYTHON3 [SCRIPTNAME]'
###############################################

#sudo python3 /home/pi/Documents/BluetoothExample/espresso-ble-
master/Reuben_app_LED.py

import logging

import dbus
import dbus.exceptions
import dbus.mainloop.glib
import dbus.service

import threading
import time

import RPi.GPIO as GPIO

#import Adafruit_DHT
from gps import *

from rpi_ws281x import PixelStrip, Color
import argparse

from mfrc522 import SimpleMFRC522

from gpiozero import DigitalOutputDevice, MCP3008, Button
from guizero import App, PushButton, Box, Text, Picture

from ble import (
    Advertisement,
    Characteristic,
```

```
    Service,
    Application,
    find_adapter,
    Descriptor,
    Agent,
)

import struct
import requests
import array
from enum import Enum

import base64
import os

import sys
from subprocess import call

MainLoop = None
try:
    from gi.repository import GLib

    MainLoop = GLib.MainLoop
except ImportError:
    import gobject as GObject

    MainLoop = GObject.MainLoop

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)
logHandler = logging.StreamHandler()
filelogHandler = logging.FileHandler("logs.log")
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s -
%(message)s")
logHandler.setFormatter(formatter)
filelogHandler.setFormatter(formatter)
logger.addHandler(filelogHandler)
logger.addHandler(logHandler)


#VivaldiBaseUrl = "XXXXXXXXXXXX"

mainloop = None

BLUEZ_SERVICE_NAME = "org.bluez"
GATT_MANAGER_IFACE = "org.bluez.GattManager1"
LE_ADVERTISEMENT_IFACE = "org.bluez.LEAdvertisement1"
LE_ADVERTISING_MANAGER_IFACE = "org.bluez.LEAdvertisingManager1"

#DHT_SENSOR = Adafruit_DHT.DHT22
```

```
DHT1_PIN = 4
DHT2_PIN = 17

# On Cooler
TEMP_ONE_SENSOR_ADDRESS = "28-000000030f3a"
TEMP_TWO_SENSOR_ADDRESS = "28-000000039c16"

# On Spare Pi
#TEMP_ONE_SENSOR_ADDRESS = "28-000000037e68"
#TEMP_TWO_SENSOR_ADDRESS = "28-000000031c3b"

MAGNET_PIN = 0
magnet_sensor = Button(MAGNET_PIN)

light_sensor_input = MCP3008(channel=0, device=1)
battery_level_input = MCP3008(channel=1, device=1)
reader = SimpleMFRC522()

#CS1_PIN = 7
#CS2_PIN = 8
#chip_select1 = DigitalOutputDevice(CS1_PIN)



# LED strip configuration:
LED_COUNT = 68        # Number of LED pixels.
LED_PIN = 18          # GPIO pin connected to the pixels (18 uses PWM!).
# LED_PIN = 10        # GPIO pin connected to the pixels (10 uses SPI /dev/spidev0.0).
LED_FREQ_HZ = 800000  # LED signal frequency in hertz (usually 800khz)
LED_DMA = 10          # DMA channel to use for generating signal (try 10)
LED_BRIGHTNESS = 255  # Set to 0 for darkest and 255 for brightest
LED_INVERT = False    # True to invert the signal (when using NPN transistor level
shift)
LED_CHANNEL = 0       # set to '1' for GPIOs 13, 19, 41, 45 or 53

# Global Variables
LED1State = 'OFF'
LED2State = 'OFF'
LED3State = 'OFF'
LED4State = 'OFF'
temp1 = 0
temp2 = 0
latlng = ''
LED8State = 'AUTO'
LED9State = 'RED'
LED10State = 'SOLID'
LED11State = 'AUTO'
LED12State = 'CELSIUS'
batteryLevel = 0.0
LEDClearState = 'ON'
```

```
LockServoState = 'OFF'
ServoComplete = False
profile = 'PROFILE_ONE'
iceSelected = 'ICE_ONE'
iceOneEnabled = False
iceTwoEnabled = False
iceOneNotification = 'OFF'
iceTwoNotification = 'OFF'

LED14State = 'OFF'
LED15State = 'OFF'

toggleSettings = True

# SERVO
#Set GPIO numbering mode
#GPIO.setmode(GPIO.BCM)

#Set pin 13 GPIO 27 as an output, and set servo 1 as pin 13 as PWM
GPIO.setup(27,GPIO.OUT)
servo1 = GPIO.PWM(27,50) #Note 13 is pin, 50 = 50kHz
'''
#start PWM running, but with value of 0 (pulse off)
servo1.start(0)
print("Waiting 2 seconds")
time.sleep(2)

#Let's move teh servo
print("Rotating 180 deg in 10 steps")

#Define variable duty
duty = 2

#Loop for duty values from 2 to 12 (0 to 180 deg)
while duty <= 12:
    servo1.ChangeDutyCycle(duty)
    time.sleep(1)
    duty = duty + 1

#Wait for a couple seconds
time.sleep(2)

#Turn back to 90 deg
print("Turning back to 90 deg for 2 seconds")
servo1.ChangeDutyCycle(7)
time.sleep(2)

#Turn back to 0 deg
print("Turning back to 0 deg")
servo1.ChangeDutyCycle(2)
```

```
time.sleep(0.5)
servo1.ChangeDutyCycle(0)

#Clean things up at the end
servo1.stop()
#GPIO.cleanup()
print("Goodbye")
'''

pin1 = DigitalOutputDevice(20)
pin2 = DigitalOutputDevice(21)
pin3 = DigitalOutputDevice(16)
pin4 = DigitalOutputDevice(6)
#pin5 = DigitalOutputDevice(6)
#pin6 = DigitalOutputDevice(13)

def startLED1():
    global LED1State
    LED1State = "ON"
    logger.debug("Running Start LED")
    start_button1.disable()
    logger.debug("Start button disable")
    stop_button1.enable()
    logger.debug("Start button enable")
    pin1.on()

def stopLED1():
    global LED1State
    LED1State = "OFF"
    logger.debug("Running Stop LED")
    start_button1.enable()
    stop_button1.disable()
    pin1.off()

def startLED2():
    global LED2State
    LED2State = "ON"
    logger.debug("Running Start LED")
    start_button2.disable()
    logger.debug("Start button disable")
    stop_button2.enable()
    logger.debug("Start button enable")
    runServo0()
    #pin2.on()

def stopLED2():
    global LED2State
    LED2State = "OFF"
    logger.debug("Running Stop LED")
    start_button2.enable()
```

```
    stop_button2.disable()
    runServo90()
    #pin2.off()

def startLED3():
    global LED3State
    global LockServoState
    global ServoComplete

    if(magnet_sensor.is_pressed):
        LED3State = "ON"
        logger.debug("Running Start LED")
        start_button3.disable()
        logger.debug("Start button disable")
        stop_button3.enable()
        logger.debug("Start button enable")
        runServo0()
        LockServoState = "ON"
        ServoComplete = True
    #pin3.on()


def stopLED3():
    global LED3State
    global LockServoState
    global ServoComplete

    LED3State = "OFF"
    logger.debug("Running Stop LED")
    start_button3.enable()
    stop_button3.disable()

    LockServoState = "OFF"
    runServo90()
    ServoComplete = False
    if LED11State is 'AUTO':
        time.sleep(3)
    #pin3.off()

def startLED4():
    global LED4State
    global LEDClearState
    LED4State = "ON"
    LEDClearState = "OFF"
    logger.debug("Running Start LED")
    start_button4.disable()
    logger.debug("Start button disable")
    stop_button4.enable()
    logger.debug("Start button enable")
    LEDStripColor()
```

```
    print('Test')
    #pin4.on()

def stopLED4():
    global LED4State
    global LEDClearState
    LED4State = "OFF"
    LEDClearState = "ON"
    logger.debug("Running Stop LED")
    start_button4.enable()
    stop_button4.disable()
    LEDStripColor()

    #pin4.off()

def goToSettings():
    global toggleSettings
    logger.debug("Settings Screen")
    if(toggleSettings):
        settingsButton.text = "Back"
        MainScreenBox.hide()
        SettingsTwoScreenBox.hide()
        IceScreenBox.hide()
        SettingsScreenBox.show()
    else:
        settingsButton.text = "Settings"
        SettingsScreenBox.hide()
        SettingsTwoScreenBox.hide()
        IceScreenBox.hide()
        MainScreenBox.show()
    toggleSettings = not toggleSettings

def goToSettingsTwo():
    SettingsScreenBox.hide()
    SettingsTwoScreenBox.show()

def goBackToSettings():
    SettingsTwoScreenBox.hide()
    SettingsScreenBox.show()

def powerOff():
    saveSettings('/home/pi/Documents/BluetoothExample/espresso-ble-
master/savesettings')
    call("sudo shutdown -h now", shell=True)

def autoLED5():
    global LED8State
    LED8State = "AUTO"
    logger.debug("Set LED to AUTO")
    auto_button1.disable()
```

```
    manual_button1.enable()

def manualLED5():
    global LED8State
    LED8State = "MANUAL"
    logger.debug("Set LED to MANUAL")
    manual_button1.disable()
    auto_button1.enable()

def redLED6():
    global LED9State
    LED9State = "RED"
    logger.debug("Set color to RED")
    red_button.disable()
    blue_button.enable()
    white_button.enable()
    LEDStripColor()
    #rainbow_button1.enable()

def blueLED6():
    global LED9State
    LED9State = "BLUE"
    logger.debug("Set color to BLUE")
    red_button.enable()
    blue_button.disable()
    white_button.enable()
    LEDStripColor()
    #rainbow_button1.enable()

def whiteLED6():
    global LED9State
    LED9State = "WHITE"
    logger.debug("Set color to WHITE")
    red_button.enable()
    blue_button.enable()
    white_button.disable()
    LEDStripColor()
    #rainbow_button1.enable()

'''
def rainbowLED6():
    global LED9State
    LED9State = "RAINBOW"
    logger.debug("Set color to RAINBOW")
    red_button.enable()
    blue_button.enable()
    white_button.enable()
    rainbow_button1.disable()
'''
```

```
def solidLED7():
    global LED10State
    LED10State = "SOLID"
    logger.debug("Set LED to SOLID")
    solid_button.disable()
    #pulse_button.enable()
    rainbow_button2.enable()
    if LED9State is 'RED':
        red_button.disable()
        blue_button.enable()
        white_button.enable()
    elif LED9State is 'BLUE':
        red_button.enable()
        blue_button.disable()
        white_button.enable()
    else:
        red_button.enable()
        blue_button.enable()
        white_button.disable()
    LEDStripColor()

def rainbowLED7():
    global LED10State
    LED10State = "RAINBOW"
    logger.debug("Set LED to RAINBOW")
    solid_button.enable()
    #pulse_button.enable()
    rainbow_button2.disable()
    red_button.disable()
    blue_button.disable()
    white_button.disable()
    LEDStripColor()

def autoLED8():
    global LED11State
    LED11State = "AUTO"
    logger.debug("Set lock to AUTO")
    auto_button2.disable()
    manual_button2.enable()

def manualLED8():
    global LED11State
    LED11State = "MANUAL"
    logger.debug("Set lock to MANUAL")
    auto_button2.enable()
    manual_button2.disable()

def celsiusLED9():
    global LED12State
    LED12State = "CELSIUS"
```

```
    logger.debug("Set temp to CELSIUS")
    celsius_button.disable()
    fahrenheit_button.enable()

def fahrenheitLED9():
    global LED12State
    LED12State = "FAHRENHEIT"
    logger.debug("Set temp to FAHRENHEIT")
    celsius_button.enable()
    fahrenheit_button.disable()

def profile1LED10():
    global profile
    profile = "PROFILE_ONE"
    profile1_button.disable()
    profile2_button.enable()

def profile2LED10():
    global profile
    profile = "PROFILE_TWO"
    profile2_button.disable()
    profile1_button.enable()

def saveSettings(filename):
    with open(filename, 'w') as f:
        s = f"""SPEAKER={LED1State}
GPS={LED2State}
LOCK={LED3State}
LED={LED4State}
TEMP1={temp1}
TEMP2={temp2}
LATLNG={latlng}
LEDMODE={LED8State}
LEDCOLOR={LED9State}
LEDEFFECT={LED10State}
LOCKMODE={LED11State}
TEMPUNITS={LED12State}
BATTERY={batteryLevel}
LEDCLEAR={LEDClearState}
LOCKSERVOSTATE={LockServoState}
SERVOCOMPLETE={ServoComplete}"""
        f.write(s)

def profileSave():
    global profile
    if profile == "PROFILE_ONE":
        saveSettings('/home/pi/Documents/BluetoothExample/espresso-ble-
master/profileonesavesettings')
    else:
```

```
        saveSettings('/home/pi/Documents/BluetoothExample/espresso-ble-
master/profiletwosavesettings')

def profileSet():
    global profile
    if profile == 'PROFILE_ONE':
        with open('/home/pi/Documents/BluetoothExample/espresso-ble-
master/profileonesavesettings','r') as f:
            contents = f.read()
            setSettings(contents)
    else:
        with open('/home/pi/Documents/BluetoothExample/espresso-ble-
master/profiletwosavesettings','r') as f:
            contents = f.read()
            setSettings(contents)

def ice1LED11():
    global iceSelected
    iceSelected = 'ICE_ONE'

def ice2LED11():
    global iceSelected
    iceSelected = 'ICE_TWO'

def iceSet():
    global iceSelected
    global iceOneEnabled
    global iceTwoEnabled
    if iceSelected == 'ICE_ONE':
        iceOneEnabled = True
        ice1_button.bg = GREEN
    else:
        iceTwoEnabled = True
        ice2_button.bg = GREEN

def iceClear():
    global iceSelected
    global iceOneEnabled
    global iceTwoEnabled
    global iceOneNotification
    global iceTwoNotification
    if iceSelected == 'ICE_ONE':
        iceOneEnabled = False
        iceOneNotification = 'OFF'
        ice1_button.bg = RED
    else:
        iceTwoEnabled = False
        iceTwoNotification = 'OFF'
        ice2_button.bg = RED
```

```
def showIce(comp):
    global iceSelected
    global iceOneNotification
    global iceTwoNotification
    SettingsScreenBox.hide()
    SettingsTwoScreenBox.hide()
    MainScreenBox.hide()
    if comp == 'one':
        iceSelected = 'ICE_ONE'
        iceOneNotification = 'ON'
    else:
        iceSelected = 'ICE_TWO'
        iceTwoNotification = 'ON'
    text14.value = f"""Ice needs to be replaced in compartment {comp}"""
    IceScreenBox.show()

def closeIce():
    IceScreenBox.hide()
    SettingsScreenBox.hide()
    SettingsTwoScreenBox.hide()
    MainScreenBox.show()
    iceClear()


app = App(width=1000, height=600, layout="auto")

TURQUOISE = "#55D6C2"
SALMONPINK = "#F49097"
OFFWHITE = "#F2F5FF"
CHARLESTONGREEN = "#222725"
DARKELECTRICBLUE = "#546A7B"
JET = "#353535"
INDIGODYE = "#284B63"
GAINSBORO = "#D9D9D9"

titleBoxBgColor = INDIGODYE
titleTextColor = OFFWHITE
titleFontSize = 60

deviceBoxBgColor = GAINSBORO
deviceTextColor = CHARLESTONGREEN
deviceFontSize = 20

tempBoxBgColor = JET
tempTextColor = OFFWHITE

deviceBox1Width = 50
deviceBox2Width = 150
deviceVerticalSpace = 2
```

```
TitleBox = Box(app, width="fill", height=100)
TitleBox.bg = titleBoxBgColor
batteryIndicator = Picture(TitleBox,
image="/home/pi/Documents/BluetoothExample/espresso-ble-
master/NotCharging1.png", align="left")
powerButton = PushButton(TitleBox, command=powerOff, text="Power", width=5,
height="fill", align="left")
#batteryIndicator2 = Picture(TitleBox,
image="/home/pi/Documents/BluetoothExample/espresso-ble-
master/NotCharging1.png", align="left")
textTitle = Text(TitleBox, text="The Smart Cooler", width="fill", height="fill",
size=titleFontSize, font="Helvetica", color=titleTextColor, align="left")
settingsButton = PushButton(TitleBox, command=goToSettings, text="Settings",
width=5, height="fill", align="left")

MainScreenBox = Box(app, width="fill", height="fill")

LED1Box = Box(MainScreenBox, width="fill", height=80)
LED1Box.bg = deviceBoxBgColor
text1 = Text(LED1Box, text="Speakers", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
start_button1 = PushButton(LED1Box, command=startLED1, text="On", width=5,
align="left")
start_button1.text_size = deviceFontSize
start_button1.text_color = deviceTextColor
spaceBox1 = Box(LED1Box, width=deviceBox1Width, align="left")
stop_button1 = PushButton(LED1Box, command=stopLED1, text="Off", enabled=False,
width=5, align="left")
stop_button1.text_size = deviceFontSize
stop_button1.text_color = deviceTextColor
spaceBox2 = Box(LED1Box, width=deviceBox2Width, align="left")

verticalSpaceBox1 = Box(MainScreenBox, width="fill", height=deviceVerticalSpace)
verticalSpaceBox1.bg = deviceBoxBgColor

LED2Box = Box(MainScreenBox, width="fill", height=80)
LED2Box.bg = deviceBoxBgColor
text2 = Text(LED2Box, text="GPS", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
start_button2 = PushButton(LED2Box, command=startLED2, text="On", width=5,
align="left")
start_button2.text_size = deviceFontSize
spaceBox3 = Box(LED2Box, width=deviceBox1Width, align="left")
stop_button2 = PushButton(LED2Box, command=stopLED2, text="Off", enabled=False,
width=5, align="left")
stop_button2.text_size = deviceFontSize
spaceBox4 = Box(LED2Box, width=deviceBox2Width, align="left")

verticalSpaceBox2 = Box(MainScreenBox, width="fill", height=deviceVerticalSpace)
verticalSpaceBox2.bg = deviceBoxBgColor
```

```
LED3Box = Box(MainScreenBox, width="fill", height=80)
LED3Box.bg = deviceBoxBgColor
text3 = Text(LED3Box, text="Lock", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
start_button3 = PushButton(LED3Box, command=startLED3, text="On", width=5,
align="left")
start_button3.text_size = deviceFontSize
spaceBox5 = Box(LED3Box, width=deviceBox1Width, align="left")
stop_button3 = PushButton(LED3Box, command=stopLED3, text="Off", enabled=False,
width=5, align="left")
stop_button3.text_size = deviceFontSize
spaceBox6 = Box(LED3Box, width=deviceBox2Width, align="left")

verticalSpaceBox3 = Box(MainScreenBox, width="fill", height=deviceVerticalSpace)
verticalSpaceBox3.bg = deviceBoxBgColor

LED4Box = Box(MainScreenBox, width="fill", height=80)
LED4Box.bg = deviceBoxBgColor
text4 = Text(LED4Box, text="LED", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
start_button4 = PushButton(LED4Box, command=startLED4, text="On", width=5,
align="left")
start_button4.text_size = deviceFontSize
spaceBox7 = Box(LED4Box, width=deviceBox1Width, align="left")
stop_button4 = PushButton(LED4Box, command=stopLED4, text="Off", enabled=False,
width=5, align="left")
stop_button4.text_size = deviceFontSize
spaceBox8 = Box(LED4Box, width=deviceBox2Width, align="left")

TempTitleBox = Box(MainScreenBox, width="fill", height=70)
TempTitleBox.bg = tempBoxBgColor
textTempTitle = Text(TempTitleBox, text="Compartment Temperature", size=24,
color=tempTextColor, width="fill", align="bottom")

TempBox = Box(MainScreenBox, width="fill", height=150)
TempBox.bg = tempBoxBgColor
spaceBox = Box(TempBox, width=60, align="left")
textTemp1 = Text(TempBox, text="Temp 1:", size=30, width=10, color=tempTextColor,
align="left")
textTemp1Reading = Text(TempBox, text="00.0", size=30, width=10,
color=tempTextColor, align="left")
textTemp2 = Text(TempBox, text="Temp 2:", size=30, width=10, color=tempTextColor,
align="left")
textTemp2Reading = Text(TempBox, text="00.0", size=30, width=10,
color=tempTextColor, align="left")

SettingsScreenBox = Box(app, width="fill", height="fill")
SettingsScreenBox.hide()
```

```
deviceBoxWidth = 80
buttonFontSize = 12
buttonWidth = 12

deviceBox1Width = 20
deviceBox2Width = 500
deviceBox3Width = 335
deviceBox4Width = 170

LED5Box = Box(SettingsScreenBox, width="fill", height=deviceBoxWidth)
LED5Box.bg = deviceBoxBgColor
text5 = Text(LED5Box, text="LED", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
auto_button1 = PushButton(LED5Box, command=autoLED5, text="Auto",
enabled=False, width=buttonWidth, align="left")
auto_button1.text_size = buttonFontSize
auto_button1.text_color = deviceTextColor
spaceBox9 = Box(LED5Box, width=deviceBox1Width, align="left")
manual_button1 = PushButton(LED5Box, command=manualLED5, text="Manual",
width=buttonWidth, align="left")
manual_button1.text_size = buttonFontSize
manual_button1.text_color = deviceTextColor
spaceBox10 = Box(LED5Box, width=deviceBox2Width, align="left")

LED6Box = Box(SettingsScreenBox, width="fill", height=deviceBoxWidth)
LED6Box.bg = deviceBoxBgColor
text6 = Text(LED6Box, text="LED Color", width="fill", align="left",
size=deviceFontSize, color=deviceTextColor)
red_button = PushButton(LED6Box, command=redLED6, text="Red", enabled=False,
width=buttonWidth, align="left")
red_button.text_size = buttonFontSize
red_button.text_color = deviceTextColor
spaceBox11 = Box(LED6Box, width=deviceBox1Width, align="left")
blue_button = PushButton(LED6Box, command=blueLED6, text="Blue",
width=buttonWidth, align="left")
blue_button.text_size = buttonFontSize
blue_button.text_color = deviceTextColor
spaceBox12 = Box(LED6Box, width=deviceBox1Width, align="left")
white_button = PushButton(LED6Box, command=whiteLED6, text="White",
width=buttonWidth, align="left")
white_button.text_size = buttonFontSize
white_button.text_color = deviceTextColor
spaceBox13 = Box(LED6Box, width=deviceBox3Width, align="left")
'''
rainbow_button1 = PushButton(LED6Box, command=rainbowLED6, text="Rainbow",
width=buttonWidth, align="left")
rainbow_button1.text_size = buttonFontSize
rainbow_button1.text_color = deviceTextColor
spaceBox14 = Box(LED6Box, width=deviceBox4Width, align="left")
'''
```

```
LED7Box = Box(SettingsScreenBox, width="fill", height=deviceBoxWidth)
LED7Box.bg = deviceBoxBgColor
text7 = Text(LED7Box, text="LED Effect", width="fill", align="left",
size=deviceFontSize, color=deviceTextColor)
solid_button = PushButton(LED7Box, command=solidLED7, text="Solid",
enabled=False, width=buttonWidth, align="left")
solid_button.text_size = buttonFontSize
solid_button.text_color = deviceTextColor
spaceBox15 = Box(LED7Box, width=deviceBox1Width, align="left")
'''
pulse_button = PushButton(LED7Box, command=pulseLED7, text="Pulse",
width=buttonWidth, align="left")
pulse_button.text_size = buttonFontSize
pulse_button.text_color = deviceTextColor
spaceBox16 = Box(LED7Box, width=deviceBox1Width, align="left")
'''
rainbow_button2 = PushButton(LED7Box, command=rainbowLED7, text="Rainbow",
width=buttonWidth, align="left")
rainbow_button2.text_size = buttonFontSize
rainbow_button2.text_color = deviceTextColor
spaceBox17 = Box(LED7Box, width=deviceBox2Width, align="left")

LED8Box = Box(SettingsScreenBox, width="fill", height=deviceBoxWidth)
LED8Box.bg = deviceBoxBgColor
text8 = Text(LED8Box, text="Lock", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
auto_button2 = PushButton(LED8Box, command=autoLED8, text="Auto",
enabled=False, width=buttonWidth, align="left")
auto_button2.text_size = buttonFontSize
auto_button2.text_color = deviceTextColor
spaceBox18 = Box(LED8Box, width=deviceBox1Width, align="left")
manual_button2 = PushButton(LED8Box, command=manualLED8, text="Manual",
width=buttonWidth, align="left")
manual_button2.text_size = buttonFontSize
manual_button2.text_color = deviceTextColor
spaceBox19 = Box(LED8Box, width=deviceBox2Width, align="left")

LED10Box = Box(SettingsScreenBox, width="fill", height=deviceBoxWidth)
LED10Box.bg = deviceBoxBgColor
text10 = Text(LED10Box, text="More", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
settings_two_button = PushButton(LED10Box, command=goToSettingsTwo,
text="Next", width=buttonWidth, align="left")
settings_two_button.text_size = buttonFontSize
settings_two_button.text_color = deviceTextColor
spaceBox21 = Box(LED10Box, width=670, align="left")

SettingsTwoScreenBox = Box(app, width="fill", height="fill")
SettingsTwoScreenBox.hide()
```

```
LED9Box = Box(SettingsTwoScreenBox, width="fill", height=deviceBoxWidth)
LED9Box.bg = deviceBoxBgColor
text9 = Text(LED9Box, text="Temp", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
celsius_button = PushButton(LED9Box, command=celsiusLED9, text="Celsius",
enabled=False, width=buttonWidth, align="left")
celsius_button.text_size = buttonFontSize
celsius_button.text_color = deviceTextColor
spaceBox19 = Box(LED9Box, width=deviceBox1Width, align="left")
fahrenheit_button = PushButton(LED9Box, command=fahrenheitLED9,
text="Fahrenheit", width=buttonWidth, align="left")
fahrenheit_button.text_size = buttonFontSize
fahrenheit_button.text_color = deviceTextColor
spaceBox20 = Box(LED9Box, width=deviceBox2Width, align="left")

LED12Box = Box(SettingsTwoScreenBox, width="fill", height=deviceBoxWidth)
LED12Box.bg = deviceBoxBgColor
text12 = Text(LED12Box, text="Profiles", width="fill", align="left",
size=deviceFontSize, color=deviceTextColor)
profile1_button = PushButton(LED12Box, command=profile1LED10, text="Profile 1",
enabled=False, width=buttonWidth, align="left")
profile1_button.text_size = buttonFontSize
profile1_button.text_color = deviceTextColor
spaceBox23 = Box(LED12Box, width=deviceBox1Width, align="left")
profile2_button = PushButton(LED12Box, command=profile2LED10, text="Profile 2",
width=buttonWidth, align="left")
profile2_button.text_size = buttonFontSize
profile2_button.text_color = deviceTextColor
spaceBox24 = Box(LED12Box, width=deviceBox4Width, align="left")
psave_button = PushButton(LED12Box, command=profileSave, text="Save",
width=buttonWidth, align="left")
psave_button.text_size = buttonFontSize
psave_button.text_color = deviceTextColor
spaceBox25 = Box(LED12Box, width=deviceBox1Width, align="left")
pset_button = PushButton(LED12Box, command=profileSet, text="Set",
width=buttonWidth, align="left")
pset_button.text_size = buttonFontSize
pset_button.text_color = deviceTextColor
spaceBox26 = Box(LED12Box, width=deviceBox1Width, align="left")

WHITE = '#FFFFFF'
RED = '#FF0000'
GREEN = '#3CB371'

LED13Box = Box(SettingsTwoScreenBox, width="fill", height=deviceBoxWidth)
LED13Box.bg = deviceBoxBgColor
text13 = Text(LED13Box, text="Ice Monitor", width="fill", align="left",
size=deviceFontSize, color=deviceTextColor)
```

```
ice1_button = PushButton(LED13Box, command=ice1LED11, text="Comp 1",
width=buttonWidth, align="left")
ice1_button.text_size = buttonFontSize
ice1_button.text_color = WHITE
ice1_button.bg = RED
spaceBox27 = Box(LED13Box, width=deviceBox1Width, align="left")
ice2_button = PushButton(LED13Box, command=ice2LED11, text="Comp 2",
width=buttonWidth, align="left")
ice2_button.text_size = buttonFontSize
ice2_button.text_color = WHITE
ice2_button.bg = RED
spaceBox28 = Box(LED13Box, width=deviceBox4Width, align="left")
iceset_button = PushButton(LED13Box, command=iceSet, text="Set",
width=buttonWidth, align="left")
iceset_button.text_size = buttonFontSize
iceset_button.text_color = deviceTextColor
spaceBox29 = Box(LED13Box, width=deviceBox1Width, align="left")
iceclear_button = PushButton(LED13Box, command=iceClear, text="Clear",
width=buttonWidth, align="left")
iceclear_button.text_size = buttonFontSize
iceclear_button.text_color = deviceTextColor
spaceBox30 = Box(LED13Box, width=deviceBox1Width, align="left")

LED11Box = Box(SettingsTwoScreenBox, width="fill", height=deviceBoxWidth)
LED11Box.bg = deviceBoxBgColor
text11 = Text(LED11Box, text="Back", width="fill", align="left", size=deviceFontSize,
color=deviceTextColor)
settings_two_button = PushButton(LED11Box, command=goBackToSettings,
text="Back", width=buttonWidth, align="left")
settings_two_button.text_size = buttonFontSize
settings_two_button.text_color = deviceTextColor
spaceBox22 = Box(LED11Box, width=670, align="left")

IceScreenBox = Box(app, width="fill", height="fill")
IceScreenBox.hide()

spaceBox31 = Box(IceScreenBox, width=deviceBox1Width, height=100)
text14 = Text(IceScreenBox, text="Ice needs to be replaced", size=deviceFontSize,
color=deviceTextColor)
spaceBox32 = Box(IceScreenBox, width=deviceBox1Width, height=100)
ice_close_button = PushButton(IceScreenBox, command=closeIce, text="OK",
width=buttonWidth)
ice_close_button.text_size = buttonFontSize
ice_close_button.text_color = deviceTextColor


def runServo():
    global servo1
    global ServoComplete
```

```
    while True:
        #start PWM running, but with value of 0 (pulse off)
        #servo1.start(0)

        if LockServoState is 'ON':
            if ServoComplete is False:
                pin3.on()
                servo1.start(0)
                print("Waiting 2 seconds")
                servo1.ChangeDutyCycle(12)
                ServoComplete = True
                time.sleep(2)
                pin3.off()
            time.sleep(1)
        else:
            #Turn back to 0 deg
            if ServoComplete is False:
                pin3.on()
                print("Turning back to 0 deg")
                servo1.ChangeDutyCycle(2)
                time.sleep(0.5)
                servo1.ChangeDutyCycle(0)
                ServoComplete = True
                time.sleep(2)
                pin3.off()

    #Clean things up at the end
    #servo1.stop()

def runServo90():
    global servo1

    pin3.on()
    #start PWM running, but with value of 0 (pulse off)
    print("Turning to 90 deg")
    time.sleep(1)
    servo1.start(0)
    servo1.ChangeDutyCycle(6)
    time.sleep(1)
    pin3.off()

def runServo0():
    global servo1

    pin3.on()
    time.sleep(1)
    print("Turning back to 0 deg")
    servo1.ChangeDutyCycle(2)
    time.sleep(0.5)
```

```
    servo1.ChangeDutyCycle(0)
    time.sleep(1)
    pin3.off()



# Define functions which animate LEDs in various ways.
def colorSolid(strip, color):
    for i in range(strip.numPixels()):
        strip.setPixelColor(i, color)
    strip.show()

def wheel(pos):
    """Generate rainbow colors across 0-255 positions."""
    if pos < 85:
        return Color(pos * 3, 255 - pos * 3, 0)
    elif pos < 170:
        pos -= 85
        return Color(255 - pos * 3, 0, pos * 3)
    else:
        pos -= 170
        return Color(0, pos * 3, 255 - pos * 3)

def rainbowSolid(strip, wait_ms=20, iterations=1):
    """Draw rainbow that fades across all pixels at once."""
    for j in range(256 * iterations):
        for i in range(strip.numPixels()):
            strip.setPixelColor(i, wheel((i + j) & 255))
    strip.show()
        #time.sleep(wait_ms / 1000.0)

def rainbow(strip, wait_ms=20, iterations=1):
    """Draw rainbow that fades across all pixels at once."""
    for j in range(256 * iterations):
        for i in range(strip.numPixels()):
            strip.setPixelColor(i, wheel((i + j) & 255))
        strip.show()
        time.sleep(wait_ms / 1000.0)

def LEDStripColor():
    global LED9State
    global LED10State
    global LEDClearState

    if LEDClearState is 'OFF':
        if LED10State is 'SOLID':
            if LED9State is "RED":
                LEDcolor = Color(255, 0, 0)
            elif LED9State is "BLUE":
                LEDcolor = Color(0, 0, 255)
```

```
            else:
                LEDcolor = Color(100, 100, 100)
            colorSolid(strip, LEDcolor)
        else:
            rainbowSolid(strip)
    else:
        LEDcolor = Color(0, 0, 0)
        colorSolid(strip,LEDcolor)

#############################################
# TO RUN LEDS, RUN PYTHON SCRIPT IN 'SUDO PYTHON3 [SCRIPTNAME]'
#############################################


# Create NeoPixel object with appropriate configuration.
strip = PixelStrip(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA,
LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL)
# Intialize the library (must be called once before other functions).
strip.begin()
LEDcolor = Color(0, 0, 0)
colorSolid(strip,LEDcolor)


# Main program logic follows:
def runLED():
    # Process arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--clear', action='store_true', help='clear the display on exit')
    args = parser.parse_args()

    # Create NeoPixel object with appropriate configuration.
    strip = PixelStrip(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA,
LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL)
    # Intialize the library (must be called once before other functions).
    strip.begin()

    print('Press Ctrl-C to quit.')
    if not args.clear:
        print('Use "-c" argument to clear LEDs on exit')

    try:

        while True:
            #rainbow(strip)
            if LEDClearState is "ON":
                LEDcolor = Color(0, 0, 0)
                colorSolid(strip,LEDcolor)
            else:
                if LED10State is "SOLID":
                    if LED9State is "RED":
```

```
                LEDcolor = Color(255, 0, 0)
            elif LED9State is "BLUE":
                LEDcolor = Color(0, 0, 255)
            else:
                LEDcolor = Color(100, 100, 100)
            colorSolid(strip, LEDcolor)
        elif LED10State is "PULSE":
            if LED9State is "RED":
                colorWipe(strip, Color(255, 0, 0))
                colorWipe(strip, Color(100, 0, 0))
            elif LED9State is "BLUE":
                colorWipe(strip, Color(0, 0, 255))
                colorWipe(strip, Color(0, 0, 100))
            else:
                colorWipe(strip, Color(100, 100, 100))
                colorWipe(strip, Color(20, 20, 20))
        else:
            rainbow(strip)


    except KeyboardInterrupt:
        if args.clear:
            colorWipe(strip, Color(0, 0, 0), 10)


def getPositionData(gps):

    # First, open terimal and run
    # sudo systemctl stop gpsd.socket
    # sudo gpsd /dev/serial0 -F /var/run/gpsd.sock

    global latlng

    nx = gpsd.next()
    # For a list of all supported classes and fields refer to:
    # https://gpsd.gitlab.io/gpsd/gpsd_json.html
    print("GPS data called")
    if nx['class'] == 'TPV':
        latitude = getattr(nx,'lat', "Unknown")
        longitude = getattr(nx,'lon', "Unknown")

        if(latitude != "Unknown" or longitude != "Unknown"):
            lng = round(longitude, 6)
            lat = round(latitude, 6)
            latlng = (str(lat) + ',' + str(lng))

        print ("Your position: lon = " + str(longitude) + ", lat = " + str(latitude))

gpsd = gps(mode=WATCH_ENABLE|WATCH_NEWSTYLE)
```

```
def readTempSensor(address):
    '''
    Initial startup
    sudo nano /boot/config.txt
    Scroll to bottom and put
    dtoverlay=w1-gpio
    then save and reboot, then type
    ls -l /sys/bus/w1/devices/
    to see device addresses
    Example code to see data being read
    cat /sys/bus/w1/devices/28-000000037e68/w1_slave
    '''
    tempfile = open("/sys/bus/w1/devices/" + address + "/w1_slave")
    thetext = tempfile.read()
    tempfile.close()
    temptextsplittest = thetext.split("\n")
    if len(temptextsplittest) > 1:
        temptextsplit = thetext.split("\n")[1].split(" ")
        if (len(temptextsplit)>9):
            print("TEMP READ")
            tempdata = temptextsplit[9]
            temperature = float(tempdata[2:])
            temperature = temperature/1000
            return temperature
        else:
            print("TEMP NOT THERE")
            #quit()
            return 0
    else:
        print("TEMP NOT THERE")
        #quit()
        return 0



def convertCelsiusToFahrenheit(celsius):
    return (celsius * 9 / 5) + 32;

def setSettings(file):
    text = file.split('\n')
    print(text)
    for state in text:
        t = state.split('=')
        if(t[0] == 'SPEAKER'):
            if(t[1] == 'ON'):
                startLED1()
            else:
                stopLED1()
        elif(t[0] == 'GPS'):
            if(t[1] == 'ON'):
```

```
        startLED2()
    else:
        stopLED2()
elif(t[0] == 'LOCK'):
    if(t[1] == 'ON'):
        startLED3()
    else:
        stopLED3()
elif(t[0] == 'LED'):
    if(t[1] == 'ON'):
        startLED4()
    else:
        stopLED4()
elif(t[0] == 'TEMP1'):
    temp1 = t[1]
elif(t[0] == 'TEMP2'):
    temp2 = t[1]
elif(t[0] == 'LATLNG'):
    latlng = t[1]
elif(t[0] == 'LEDMODE'):
    if(t[1] == 'AUTO'):
        autoLED5()
    else:
        manualLED5()
elif(t[0] == 'LEDCOLOR'):
    if(t[1] == 'RED'):
        redLED6()
    elif(t[1] == 'BLUE'):
        blueLED6()
    else:
        whiteLED6()
elif(t[0] == 'LEDEFFECT'):
    if(t[1] == 'SOLID'):
        solidLED7()
    else:
        rainbowLED7()
elif(t[0] == 'LOCKMODE'):
    if(t[1] == 'AUTO'):
        autoLED8()
    else:
        manualLED8()
elif(t[0] == 'TEMPUNITS'):
    if(t[1] == 'CELSIUS'):
        celsiusLED9()
    else:
        fahrenheitLED9()
elif(t[0] == 'BATTERY'):
    batteryLevel = t[1]
elif(t[0] == 'LEDCLEAR'):
    LEDClearState = t[1]
```

```
        elif(t[0] == 'LOCKSERVOSTATE'):
            LockServoState = t[1]
        elif(t[0] == 'SERVOCOMPLETE'):
            ServoComplete = t[1]

########################################
# Initialize variables
########################################
with open('/home/pi/Documents/BluetoothExample/espresso-ble-master/savesettings','r')
as f:
    contents = f.read()
    setSettings(contents)

def readAnalog():

    global temp1
    global temp2
    global batteryLevel
    global batteryIndicator
    global LEDClearState
    global LED4State
    global LED8State
    global LED11State
    global ServoComplete
    global LockServoState

    while True:
        light_reading = light_sensor_input.value
        voltage = light_reading * 3.3
        print("Light Sensor Reading={:.2f}\tVoltage={:.2f}".format(light_reading, voltage))
        if magnet_sensor.is_pressed:
            print("Magnet closed")
            start_button4.disable()
            stop_button4.disable()
            LEDClearState = 'ON'
            LEDStripColor()
        else:
            # Re-enable buttons if door open
            if LED4State is 'ON':
                startLED4()
            else:
                stopLED4()


            if LED8State is 'AUTO':
                if light_reading <= 0.18:
                    LEDClearState = 'OFF'
                    LEDStripColor()
                else:
                    if LED4State is 'OFF':
```

```
                LEDClearState = 'ON'
                LEDStripColor()
        else:
            if LED4State is 'OFF':
                LEDClearState = 'ON'
                LEDStripColor()


    battery_level_reading = battery_level_input.value
    voltage = battery_level_reading * 3.3
    print("Battery Level Reading={:.2f}\tVoltage={:.2f}".format(battery_level_reading,
voltage))
    batteryLevel = battery_level_reading

    # Battery Level Indicator Logic
    if (batteryLevel <= 0.2):
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/Battery201.png"
    elif (batteryLevel > 0.2 and batteryLevel <= 0.4):
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/Battery401.png"
    elif (batteryLevel > 0.4 and batteryLevel <= 0.6):
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/Battery601.png"
    elif (batteryLevel > 0.6 and batteryLevel <= 0.8):
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/Battery801.png"
    elif (batteryLevel > 0.8):
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/Battery1001.png"
    else:
        batteryIndicator.value = "/home/pi/Documents/BluetoothExample/espresso-ble-
master/NotCharging1.png"


    temp1Celsius = readTempSensor(TEMP_ONE_SENSOR_ADDRESS)
    temp2Celsius = readTempSensor(TEMP_TWO_SENSOR_ADDRESS)
    temp1Fahrenheit = convertCelsiusToFahrenheit(temp1Celsius)
    temp2Fahrenheit = convertCelsiusToFahrenheit(temp2Celsius)

    if (LED12State == "CELSIUS"):
        temp1 = "{0:0.1f} C ".format(temp1Celsius)
        temp2 = "{0:0.1f} C ".format(temp2Celsius)
        textTemp1Reading.value = temp1
        textTemp2Reading.value = temp2
    else:
        temp1 = "{0:0.1f} F ".format(temp1Fahrenheit)
        temp2 = "{0:0.1f} F ".format(temp2Fahrenheit)
        textTemp1Reading.value = temp1
        textTemp2Reading.value = temp2
```

```
        print("Temp Sensor One: " + temp1)
        print("Temp Sensor Two: " + temp2)
        #print("Temp Sensor One: {0:0.2f}C".format(temp1))
        #print("Temp Sensor Two: {0:0.2f}C".format(temp2))

        if magnet_sensor.is_pressed:
            if LED3State is 'ON':
                start_button3.disable()
                stop_button3.enable()
            else:
                start_button3.enable()
                stop_button3.disable()

            print("Magnet closed")
            if LED11State is 'AUTO':
                if ServoComplete is False:
                    if LockServoState is 'OFF':
                        startLED3()
                        ServoComplete = True

        else:
            start_button3.disable()
            stop_button3.disable()
            ServoComplete = False

        if LED2State is 'ON':
            getPositionData(gpsd)

        if temp1Celsius >= 30.0:
            if iceOneEnabled == True:
                iceSelected = 'ICE_ONE'
                showIce('one')
        if temp2Celsius >= 30.0:
            if iceTwoEnabled == True:
                iceSelected = 'ICE_TWO'
                showIce('two')


        print(profile)
        print(f"""Ice One {iceOneEnabled}""")
        print(f"""Ice One {iceOneNotification}""")
        print(f"""Ice Two {iceTwoEnabled}""")
        print(f"""Ice Two {iceTwoNotification}""")


        time.sleep(1.5)

def readRFID():
    global LockServoState
```

```
    while True:
        print("RFID read")
        id, text = reader.read()
        print("ID: %s\nText: %s" % (id,text))
        if(magnet_sensor.is_pressed):
            #runServo()
            if LockServoState is 'ON':
                #LockServoState = "OFF"
                #runServo0()
                stopLED3()
            else:
                #runServo90()
                #LockServoState = "ON"
                startLED3()

        else:
            #LockServoState = "OFF"
            #runServo0()
            stopLED3()
        time.sleep(5)

class InvalidArgsException(dbus.exceptions.DBusException):
    _dbus_error_name = "org.freedesktop.DBus.Error.InvalidArgs"


class NotSupportedException(dbus.exceptions.DBusException):
    _dbus_error_name = "org.bluez.Error.NotSupported"


class NotPermittedException(dbus.exceptions.DBusException):
    _dbus_error_name = "org.bluez.Error.NotPermitted"


class InvalidValueLengthException(dbus.exceptions.DBusException):
    _dbus_error_name = "org.bluez.Error.InvalidValueLength"


class FailedException(dbus.exceptions.DBusException):
    _dbus_error_name = "org.bluez.Error.Failed"


def register_app_cb():
    logger.info("GATT application registered")


def register_app_error_cb(error):
    logger.critical("Failed to register application: " + str(error))
    mainloop.quit()
```

```
class SmartCoolerService(Service):
    """
    Dummy test service that provides characteristics and descriptors that
    exercise various API functionality.

    """

    COOLER_SVC_UUID = "12634d89-d598-4874-8e86-7d042ee07ba7"

    def __init__(self, bus, index):
        Service.__init__(self, bus, index, self.COOLER_SVC_UUID, True)
        self.add_characteristic(LED1(bus, 0, self))
        self.add_characteristic(LED2(bus, 1, self))
        self.add_characteristic(LED3(bus, 2, self))
        self.add_characteristic(LED4(bus, 3, self))
        self.add_characteristic(LED5(bus, 4, self))
        self.add_characteristic(LED6(bus, 5, self))
        self.add_characteristic(LED7(bus, 6, self))
        self.add_characteristic(LED8(bus, 7, self))
        self.add_characteristic(LED9(bus, 8, self))
        self.add_characteristic(LED10(bus, 9, self))
        self.add_characteristic(LED11(bus, 10, self))
        self.add_characteristic(LED12(bus, 11, self))
        self.add_characteristic(LED13(bus, 12, self))
        self.add_characteristic(LED14(bus, 13, self))
        self.add_characteristic(LED15(bus, 14, self))


class LED1(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c14e"
    description = b"Speakers ON/OFF"

    global LED1State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED1State.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Speakers Read: " + repr(self.value))
        test = bytes(LED1State.encode("utf-8"))
        self.value = test

        return self.value
```

```
    def WriteValue(self, value, options):
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)
        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        if decodedValue == "ON":
            logger.debug("Should be on")
            startLED1()
        else:
            logger.debug("Should be off")
            stopLED1()

        self.value = cmd

class LED2(Characteristic):
    uuid = "322e774f-c909-49c4-bd7b-48a4003a967f"
    description = b"LED Lights ON/OFF"

    global LED2State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED2State.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("LED Lights Read: " + repr(self.value))
        test = bytes(LED2State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)
        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        if decodedValue == "ON":
            logger.debug("Should be on")
            startLED2()
        else:
            logger.debug("Should be off")
            stopLED2()
```

```
        self.value = cmd

class LED3(Characteristic):
    uuid = "9c7dbce8-de5f-4168-89dd-74f04f4e5842"
    description = b"Lock ON/OFF"

    global LED3State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED3State.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Lock Read: " + repr(self.value))
        test = bytes(LED3State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)
        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        if decodedValue == "ON":
            logger.debug("Should be on")
            startLED3()
        else:
            logger.debug("Should be off")
            stopLED3()

        self.value = cmd


class LED4(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c144"
    description = b"GPS ON/OFF"

    global LED4State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
```

```
        )

        test = bytes(LED4State.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("GPS Read: " + repr(self.value))
        test = bytes(LED4State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)
        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        if decodedValue == "ON":
            logger.debug("Should be on")
            startLED4()
        else:
            logger.debug("Should be off")
            stopLED4()

        self.value = cmd


class LED5(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c145"
    description = b"Temp 1 Unit Reading"

    global temp1

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes("0".encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Temp 1 Read: " + repr(self.value))
        print(temp1)
        test = bytes(str(temp1).encode("utf-8"))
        self.value = test
```

```
        return self.value


class LED6(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c141"
    description = b"Temp 2 Unit Reading"

    global temp2

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes("0".encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Temp 2 Read: " + repr(self.value))
        print(temp2)
        test = bytes(str(temp2).encode("utf-8"))
        self.value = test

        return self.value


class LED7(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c142"
    description = b"GPS Location Reading"

    global latlng

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes("coords".encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Latlng Read: " + repr(self.value))
        print(latlng)
        test = bytes(str(latlng).encode("utf-8"))
        self.value = test

        return self.value
```

```
class LED8(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c150"
    description = b"LED Manual/Auto"

    global LED8State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED8State.encode("utf-8"))
        #test_base = base64.b64encode(test)
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("LED Selector Read: " + repr(self.value))
        test = bytes(LED8State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global LED8State

        logger.debug("LED 8")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        LED8State = decodedValue

        if decodedValue == "AUTO":
            logger.debug("Should be auto")
            autoLED5()
        else:
            logger.debug("Should be manual")
            manualLED5()


        self.value = cmd

class LED9(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c151"
    description = b"LED Color Selector"
```

```
    global LED9State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED9State.encode("utf-8"))
        #test_base = base64.b64encode(test)
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("LED Color Selector Read: " + repr(self.value))
        test = bytes(LED9State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global LED9State

        logger.debug("LED 9")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        LED9State = decodedValue

        if decodedValue == "RED":
            logger.debug("Should be red")
            redLED6()
        elif decodedValue == "BLUE":
            logger.debug("Should be blue")
            blueLED6()
        else:
            logger.debug("Should be white")
            whiteLED6()

        self.value = cmd

class LED10(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c152"
    description = b"LED Effect Selector"

    global LED10State
```

```
    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED10State.encode("utf-8"))
        #test_base = base64.b64encode(test)
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("LED Effect Selector Read: " + repr(self.value))
        test = bytes(LED10State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global LED10State

        logger.debug("LED 10")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        LED10State = decodedValue

        if decodedValue == "SOLID":
            logger.debug("Should be solid")
            solidLED7()
        else:
            logger.debug("Should be rainbow")
            rainbowLED7()

        self.value = cmd

class LED11(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c153"
    description = b"Lock Manual/Auto"

    global LED11State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )
```

```
        test = bytes(LED11State.encode("utf-8"))
        #test_base = base64.b64encode(test)
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Lock Selector Read: " + repr(self.value))
        test = bytes(LED11State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global LED11State

        logger.debug("LED 11")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        LED11State = decodedValue

        if decodedValue == "AUTO":
            logger.debug("Should be auto")
            autoLED8()
        else:
            logger.debug("Should be manual")
            manualLED8()

        self.value = cmd

class LED12(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c154"
    description = b"Temp Units Conversion Celsius/Fahrenheit"

    global LED12State

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(LED12State.encode("utf-8"))
        #test_base = base64.b64encode(test)
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))
```

```
    def ReadValue(self, options):
        logger.debug("Temp Conversion Units Read: " + repr(self.value))
        test = bytes(LED12State.encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global LED12State

        logger.debug("LED 12")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        LED12State = decodedValue

        if decodedValue == "CELSIUS":
            logger.debug("Should be celsius")
            celsiusLED9()
        else:
            logger.debug("Should be fahrenheit")
            fahrenheitLED9()

        self.value = cmd

class LED13(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c155"
    description = b"Battery Level Reading"

    global batteryLevel

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes("0".encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("power Read: " + repr(self.value))
        print(batteryLevel)
        test = bytes(str(batteryLevel).encode("utf-8"))
        self.value = test
```

```
        return self.value

class LED14(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c156"
    description = b"Ice Compartment One Notification"

    global iceOneNotification
    global iceSelected

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(iceOneNotification.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Ice One Notification Read: " + repr(self.value))
        print(iceOneNotification)
        test = bytes(str(iceOneNotification).encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global iceOneNotification
        global iceSelected

        logger.debug("Ice One Notification")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        iceOneNotification = decodedValue
        iceSelected = 'ICE_ONE'

        if decodedValue == "OFF":
            logger.debug("Ice One Notification Off")
            iceClear()

        self.value = cmd

class LED15(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c157"
    description = b"Ice Compartment Two Notification"
```

```
    global iceTwoNotification
    global iceSelected

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        test = bytes(iceTwoNotification.encode("utf-8"))
        self.value = test
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))

    def ReadValue(self, options):
        logger.debug("Ice Two Notification Read: " + repr(self.value))
        print(iceTwoNotification)
        test = bytes(str(iceTwoNotification).encode("utf-8"))
        self.value = test

        return self.value

    def WriteValue(self, value, options):

        global iceTwoNotification
        global iceSelected

        logger.debug("Ice Two Notification")
        logger.debug("power Write: " + repr(value))
        cmd = bytes(value)

        decodedValue = cmd.decode("utf-8")
        logger.debug("Value: " + decodedValue)

        iceTwoNotification = decodedValue
        iceSelected = 'ICE_TWO'

        if decodedValue == "OFF":
            logger.debug("Ice Two Notification Off")
            iceClear()

        self.value = cmd

class CharacteristicUserDescriptionDescriptor(Descriptor):
    """
    Writable CUD descriptor.
    """

    CUD_UUID = "2901"

    def __init__(
```

```
        self, bus, index, characteristic,
    ):

        self.value = array.array("B", characteristic.description)
        self.value = self.value.tolist()
        Descriptor.__init__(self, bus, index, self.CUD_UUID, ["read"], characteristic)

    def ReadValue(self, options):
        return self.value

    def WriteValue(self, value, options):
        if not self.writable:
            raise NotPermittedException()
        self.value = value


class SmartCoolerAdvertisement(Advertisement):
    def __init__(self, bus, index):
        Advertisement.__init__(self, bus, index, "peripheral")
        self.add_manufacturer_data(
            0xFFFF, [0x70, 0x74],
        )
        self.add_service_uuid(SmartCoolerService.COOLER_SVC_UUID)

        self.add_local_name("The Smart Cooler")
        self.include_tx_power = True


def register_ad_cb():
    logger.info("Advertisement registered")
    #os.system("bluealsa-aplay 68:E7:C2:90:28:F0")


def register_ad_error_cb(error):
    logger.critical("Failed to register advertisement: " + str(error))
    mainloop.quit()


AGENT_PATH = "/com/punchthrough/agent"


def main():
    global mainloop

    dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)

    # get the system bus
    bus = dbus.SystemBus()
    # get the ble controller
    adapter = find_adapter(bus)
```

```
if not adapter:
    logger.critical("GattManager1 interface not found")
    return

adapter_obj = bus.get_object(BLUEZ_SERVICE_NAME, adapter)

adapter_props = dbus.Interface(adapter_obj, "org.freedesktop.DBus.Properties")

# powered property on the controller to on
adapter_props.Set("org.bluez.Adapter1", "Powered", dbus.Boolean(1))

# Get manager objs
service_manager = dbus.Interface(adapter_obj, GATT_MANAGER_IFACE)
ad_manager = dbus.Interface(adapter_obj, LE_ADVERTISING_MANAGER_IFACE)

advertisement = SmartCoolerAdvertisement(bus, 0)
obj = bus.get_object(BLUEZ_SERVICE_NAME, "/org/bluez")

agent = Agent(bus, AGENT_PATH)

app = Application(bus)
app.add_service(SmartCoolerService(bus, 2))

mainloop = MainLoop()

agent_manager = dbus.Interface(obj, "org.bluez.AgentManager1")
agent_manager.RegisterAgent(AGENT_PATH, "NoInputNoOutput")

ad_manager.RegisterAdvertisement(
    advertisement.get_path(),
    {},
    reply_handler=register_ad_cb,
    error_handler=register_ad_error_cb,
)

logger.info("Registering GATT application...")

service_manager.RegisterApplication(
    app.get_path(),
    {},
    reply_handler=register_app_cb,
    error_handler=[register_app_error_cb],
)

agent_manager.RequestDefaultAgent(AGENT_PATH)
print("Is looping?")
#mainloop.run()

t3 = threading.Thread(target=readAnalog)
```

```
    t3.start()

    t4 = threading.Thread(target=runLED)
    #t4.start()

    t5 = threading.Thread(target=readRFID)
    t5.start()

    t6 = threading.Thread(target=runServo)
    #t6.start()

    t2 = threading.Thread(target=mainloop.run)
    t2.start()


    '''
    e = threading.Event()

    t1 = threading.Thread(name='GUI Interface', target=startGUI, args=(e, ))
    t1.start()
    e.set()
    '''

    # ad_manager.UnregisterAdvertisement(advertisement)
    # dbus.service.Object.remove_from_connection(advertisement)


if __name__ == "__main__":
    main()

app.display()
```

Appendix F

React Native Software Code

App.js is main program file.

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 *
 * @format
 * @flow strict-local
 */

import React, { useEffect, useState } from 'react';
import { BleManager, Device } from 'react-native-ble-plx';
import type {Node} from 'react';
import {
  SafeAreaView,
  ScrollView,
  StatusBar,
  StyleSheet,
  Text,
  useColorScheme,
  View,
  Button,
  Image,
  LogBox,
} from 'react-native';
import BluetoothDevices from './app/components/BluetoothDevices';

import {
  Colors,
  DebugInstructions,
  Header,
  LearnMoreLinks,
  ReloadInstructions,
} from 'react-native/Libraries/NewAppScreen';

LogBox.ignoreLogs(['new NativeEventEmitter']); // Ignore log notification by message
LogBox.ignoreAllLogs(); //Ignore all log notifications

let counter = 1;

const MainScreen = ({ screen, timerDone, retryScan, device }) => {
```

```
  const connectDevice = (device) => {
    console.log("Device connected. Device: " + device.id);
  };

  if (screen === 'main') {
    return (
      <View style={ styles.container }>
        <Image
          style={ styles.loading }
          source={ require('./app/static/loading.gif') }
          />

        {
          !timerDone ?
            <Text>Scanning...</Text>
            :
            <Button title="Retry Scan" onPress={ () => { retryScan() } }/>
        }


        {
          //<Button title="Start Scan" onPress={ () => { createSubscription() } } />
        }
      </View>
    )
  } else if (screen === 'connected') {
    return (
      <View style={  styles.container }>
        <BluetoothDevices
              device={ device }
              onClick={ (prop) => { connectDevice(prop) } }
              />
      </View>
    )
  } else if (screen === 'settings') {

  } else {
    return (
      <Text>Error</Text>
    )
  }
};

const App = () => {

  const [ devices, setDevices ] = React.useState([]);
  const [ timerDone, setTimerDone ] = React.useState(false);
  const [ test, setTest ] = React.useState([]);

  // Old Raspberry Pi 4 Ram 4GB
```

```
  //const RASPBERRY_PI_DEVICE_MAC_ID = 'E4:5F:01:39:36:E6';
  //const RASPBERRY_PI_DEVICE_UUID = 'FD025826-984A-1CD2-64C3-
1E24A270CCAA';

  // Raspberry Pi 8GB Ram
  const RASPBERRY_PI_DEVICE_MAC_ID = 'DC:A6:32:EC:ED:75';
  const RASPBERRY_PI_DEVICE_UUID = '87B585D1-84C3-486A-8F3D-
77CF16F84F30';
  const DEVICE_NAME = 'The Smart Cooler';

  const manager = new BleManager();

  const createSubscription = () => {

    const subscription = manager.onStateChange((state) => {
      if (state === 'PoweredOn') {
        console.log('Powered On');

        let device_list = devices;
        let scan_complete = false;

        if (devices.length == 0)


          manager.startDeviceScan(null, null, (error, device) => {
            console.log('Scanning...');

            if (error) {
              // Handle error (scanning will be stopped automatically)
              console.log('Error: ' + error)
              return
            }

            if (device.id === RASPBERRY_PI_DEVICE_MAC_ID || device.id ===
RASPBERRY_PI_DEVICE_UUID || device.localName === DEVICE_NAME) {
              device_list = [...device_list, device];
              manager.stopDeviceScan();
              console.log("final");
              console.log(device_list);
              scan_complete = true;
              setDevices(device_list);
            }



            // console.log(device);
            console.log(device.name);
            console.log(device.localName);
            console.log(device.manufacturerData);
          });
```

```
      /*
        setTimeout(() => {
          if(!scan_complete) {
            console.log('Scan complete');
            manager.stopDeviceScan();
            //setTimerDone(true);
          }
        }, 5000);
      */
      /*
      setTimeout(() => {
          manager.stopDeviceScan();
          console.log("final");
          console.log(device_list);
          setDevices(device_list);
        }, 3000);
      */

        scanAndConnect();
      subscription.remove();
    }
  }, true)
}

const scanAndConnect = () => {



  // setDevices(devices => [...devices, device.id]);
  //setTest(counter);
  counter++;
  console.log(counter);

};

const stopScan = () => {
  console.log("Stop device scan.");
  manager.stopDeviceScan();
};



useEffect( () => {
  // Anything in here will be fired on component mount.
  console.log('Component mounted.');
  // const subscription = manager.onStateChange((state) => {
  //    if (state === 'PoweredOn') {
  //        scanAndConnect();
  //        subscription.remove();
```

```
    //     }
    //  }, true);
    if (devices.length == 0) {
      createSubscription();
    }

    return () => {
      console.log("Component Unmount.");
      manager.destroy();
    };
  }, []);

  return (
    <SafeAreaView style={ styles.mainContainer }>
      { devices.length > 0 ?
        <MainScreen screen='connected' device={ devices[0] } />
        :
        <MainScreen screen='main' timerDone={ timerDone } retryScan={
createSubscription } />
      }
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  mainContainer: {
    flex: 1,
  },
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  loading: {
    width: 150,
    height: 150,
    marginBottom: 20,

  },
  sectionContainer: {
    marginTop: 32,
    paddingHorizontal: 24,
  },
  sectionTitle: {
    fontSize: 24,
    fontWeight: '600',
  },
  sectionDescription: {
    marginTop: 8,
    fontSize: 18,
```

```
    fontWeight: '400',
  },
  highlight: {
    fontWeight: '700',
  },
});

export default App;
```

BluetoothDevices.js is Bluetooth components.

```
import React, { useEffect, useState } from 'react';
import {
	View,
	Text,
	StyleSheet,
	TouchableOpacity,
	Button,
	Image,
	Switch,
	Linking,
	Pressable,
	Modal,
	Alert
} from 'react-native';
import { Base64 } from 'js-base64';
import Settings from './Settings';

const LED1_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c14e"; // Speakers ON/OFF

const LED2_UUID = "322e774f-c909-49c4-bd7b-48a4003a967f"; // GPS ON/OFF

const LED3_UUID = "9c7dbce8-de5f-4168-89dd-74f04f4e5842"; // Lock ON/OFF

const LED4_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c144"; // LED Lights ON/OFF

const LED5_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c145"; // Temp 1

const LED6_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c141"; // Temp 2

const LED7_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c142"; // GPS coordinates

const LED8_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c150"; // LED Settings

const LED9_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c151"; // LED Color Selector

const LED10_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c152"; // LED Effect Selector

const LED11_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c153"; // Lock Settings

const LED12_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c154"; // Temp Units

const LED13_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c155"; // Battery Level

const LED14_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c156"; // Ice Compartment 1
Notification
```

```
const LED15_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c157"; // Ice Compartment 2
Notification


const CharacteristicSlider = ({ onClick, characteristicState }) => {
	// const [ isEnabled, setIsEnabled ] = React.useState(false);

	let isEnabled = false;
	console.log("Slider rendered.");

	if (characteristicState == "ON") {
		isEnabled = true;
	} else {
		isEnabled = false;
	}

	const toggleSwitch = () => {
		//setIsEnabled(previousState => !previousState);
		onClick();
	};

	return (
	   <View>
	    <Switch
	      trackColor={{ false: "#767577", true: "#81b0ff" }}
	      thumbColor={isEnabled ? "#f5dd4b" : "#f4f3f4"}
	      ios_backgroundColor="#3e3e3e"
	      onValueChange={toggleSwitch}
	      value={isEnabled}
	    />
	   </View>
	 );
}

const Characteristic = ({ character }) => {

	const [ deviceCharacteristic, setDeviceCharacteristic ] = React.useState({ uuid: 1,
value: 'test' });
	const [ characteristicState, setCharacteristicState] = React.useState('OFF');
	const [ type, setType ] = React.useState(0);
	const [ featureName, setFeatureName ] = React.useState('');

	let featureNames = '';

	const setFeatureNames = (uuid) => {
		switch (uuid) {
			case LED1_UUID:
				setFeatureName('Speakers');
				break;
```

```
				case LED2_UUID:
					setFeatureName('GPS');
					break;
				case LED3_UUID:
					setFeatureName('Lock');
					break;
				case LED4_UUID:
					setFeatureName('LED Lights');
					break;
				case LED5_UUID:
					setFeatureName('Temp 1 Unit');
					break;
				case LED6_UUID:
					setFeatureName('Temp 2 Unit');
					break;
				case LED7_UUID:
					setFeatureName('GPS Location');
					break;
				default:
					setFeatureName('Unknown');
			}
		}

		const setTypeByUUID = (uuid) => {
			if (uuid == LED1_UUID || uuid == LED2_UUID || uuid == LED3_UUID
|| uuid == LED4_UUID) {
				setType(1);
			}
			else if (uuid == LED7_UUID) {
				setType(2);
			}
			else {
				setType(0);
			}
		}

		const readCharacteristic = () => {
			character.read().then((characteristic) => {

				//console.log('Characteristics');
				//console.log(characteristic);
				//console.log(characteristic.uuid);
				//console.log(characteristic.value);
				//console.log(Base64.decode(characteristic.value));

				setTypeByUUID(characteristic.uuid);
				setFeatureNames(characteristic.uuid);
				setCharacteristicState(Base64.decode(characteristic.value));
				//setDeviceCharacteristic(characteristic);
			})
```

```
		.catch((error) => {
			console.log('Catch operation: ' + error.message);
			throw error;
		});
	};

	const sendON = () => {
		let newValue = "ON";

	character.writeWithResponse(Base64.encode(newValue)).then((newCharacteristic
) => {
			console.log("New value written.");
			setCharacteristicState(newValue);
		});
	};

	const sendOFF = () => {
		let newValue = "OFF";

	character.writeWithResponse(Base64.encode(newValue)).then((newCharacteristic
) => {
			console.log("New value written.");
			setCharacteristicState(newValue);
		});
	};

	const toggleOnOff = () => {
		let newValue;
		console.log("Called toggle function");
		if (characteristicState == "OFF") {
			// Set off
			newValue = "ON";
		} else {
			newValue = "OFF";
		}


	character.writeWithResponse(Base64.encode(newValue)).then((newCharacteristic
) => {
				console.log("New value written.");
				setCharacteristicState(newValue);
			});
	};


	const googleMapsURL = 'https://www.google.com/maps/search/?api=1&query='

	const openGoogleMaps = () => {

		let latlng = characteristicState.split(',');
```

```
			let url = `${googleMapsURL}${latlng[0]}%2C${latlng[1]}`;
			console.log(url);
			Linking.canOpenURL(url).then((supported) => {
				console.log(supported);
				if (supported) {
					Linking.openURL(url);
				}
			});
		}

		useEffect( () => {
		   // Anything in here will be fired on component mount.
		   console.log('Characteristic component mounted.');
		   // const subscription = manager.onStateChange((state) => {
		   //     if (state === 'PoweredOn') {
		   //         scanAndConnect();
		   //         subscription.remove();
		   //     }
		   //  }, true);
		   readCharacteristic()
		   const readIntervals = setInterval(readCharacteristic, 3000);
		   return () => {
		     console.log("Characteristic component Unmount.");
		     //manager.destroy();
		     clearInterval(readIntervals);
		   };
		  }, []);

		return (
			<View style={styles.characteristicContainer}>
				{/* <Button onPress={ () => readCharacteristic() } title="Read" />
*/}
				{
					//<Text>{`Feature: ${deviceCharacteristic.uuid}`}</Text>
				}
					<Text
style={styles.featureNamesText}>{featureName}</Text>
				{
					type == 1 &&
						<CharacteristicSlider onClick={ toggleOnOff }
characteristicState={ characteristicState }/>
				}
				{
					type == 0 &&
						<Text style={styles.dataText}>{ characteristicState
}</Text>
				}
				{
					(type == 2 && /\d+/.test(characteristicState)) &&
```

```
                                        <Button title="View" onPress={ openGoogleMaps
}></Button>
                        }

                        { /*
                        characteristicState == 'OFF' ?
                                <Button onPress={ () => sendON() } title="ON" />
                                :
                                <Button onPress={ () => sendOFF() } title="OFF" />
                                */
                        }
                </View>
        )
}

const BatteryLevelIcon = ({ batteryCharacter }) => {
        const [ batteryLevel, setBatteryLevel ] = React.useState(0);

        let batteryLevelImageLink;

        const readBatteryCharacteristic = () => {
                console.log("Read battery");
                console.log(batteryLevel);

                batteryCharacter.read().then((characteristic) => {
                        console.log('Characteristics');
                        console.log(characteristic);
                        console.log(characteristic.uuid);
                        console.log(characteristic.value);
                        console.log(Base64.decode(characteristic.value));

                        //setTypeByUUID(characteristic.uuid);
                        //setFeatureNames(characteristic.uuid);
                        setBatteryLevel(Base64.decode(characteristic.value));
                        //setDeviceCharacteristic(characteristic);
                })
                .catch((error) => {
                        console.log('Catch operation: ' + error.message);
                        throw error;
                });

        };


        if (batteryLevel < 0.2) {
                batteryLevelImageLink = require('../static/Battery201.png');
        } else if (batteryLevel >= 0.2 && batteryLevel < 0.4) {
                batteryLevelImageLink = require('../static/Battery401.png');
        } else if (batteryLevel >= 0.4 && batteryLevel < 0.6) {
                batteryLevelImageLink = require('../static/Battery601.png');
```

```
	} else if (batteryLevel >= 0.6 && batteryLevel < 0.8) {
		batteryLevelImageLink = require('../static/Battery801.png');
	} else {
		batteryLevelImageLink = require('../static/Battery1001.png');
	}

	useEffect( () => {
	   // Anything in here will be fired on component mount.
	   console.log('Battery component mounted.');
	   // const subscription = manager.onStateChange((state) => {
	   //     if (state === 'PoweredOn') {
	   //         scanAndConnect();
	   //         subscription.remove();
	   //     }
	   //  }, true);
	   readBatteryCharacteristic()

	   const readIntervals = setInterval(readBatteryCharacteristic, 3000);
	   return () => {
	     console.log("Battery component Unmount.");
	     //manager.destroy();
	     clearInterval(readIntervals);
	   };
	  }, []);

	return (
		<View>
			<Image
				style={styles.batteryLevelIcon}
				source={ batteryLevelImageLink }
				/>
		</View>
	)
}


const IceNotification = ({ iceCharacter }) => {
	const [ ice, setIce ] = React.useState('OFF');
	const [ compartment, setCompartment ] = React.useState('ONE');
	const [ modalVisible, setModalVisible ] = useState(false);

	const setCompartmentByUUID = (uuid) => {
		if (uuid == LED14_UUID) {
			setCompartment('ONE');
		} else if (uuid == LED15_UUID) {
			setCompartment('TWO');
		}
	}

	const sendOFF = () => {
```

```
		let newValue = "OFF";

	iceCharacter.writeWithResponse(Base64.encode(newValue)).then((newCharacteri
stic) => {
			console.log("New value written.");
			//setCharacteristicState(newValue);
		});
	};

	const readIceCharacteristic = () => {
		iceCharacter.read().then((characteristic) => {
			console.log('ICE');
			console.log(Base64.decode(characteristic.value));
			setCompartmentByUUID(characteristic.uuid);
			setIce(Base64.decode(characteristic.value));
		})
	}

	useEffect( () => {
	  // Anything in here will be fired on component mount.
	  console.log('Ice component mounted.');
	  // const subscription = manager.onStateChange((state) => {
	  //   if (state === 'PoweredOn') {
	  //     scanAndConnect();
	  //     subscription.remove();
	  //   }
	  // }, true);
	  readIceCharacteristic()

	  const readIntervals = setInterval(readIceCharacteristic, 3000);
	  return () => {
	    console.log("Ice component Unmount.");
	    //manager.destroy();
	    clearInterval(readIntervals);
	  };
	}, []);

	const showNotification = () => {
		if (ice == 'ON') {
			return (
				<View style={styles.centeredView}>
			  <Modal
			    animationType="slide"
			    transparent={true}
			    visible={modalVisible}
			    onRequestClose={() => {
			      Alert.alert("Modal has been closed.");
			      setModalVisible(!modalVisible);
			    }}
			  >
```

```
                <View style={styles.centeredView}>
                  <View style={styles.modalView}>
                    <Text style={styles.modalText}>Compartment
{compartment} Ice Low.</Text>
                    <Pressable
                      style={[styles.button, styles.buttonClose]}
                      onPress={() => {
                              setModalVisible(!modalVisible)
                      }
                      }
                    >
                      <Text style={styles.textStyle}>Close</Text>
                    </Pressable>
                  </View>
                </View>
              </Modal>
              <Pressable
                style={[styles.button, styles.buttonOpen]}
                onPress={() => setModalVisible(true)}
              >
                <Text style={styles.textStyle}>Ice {compartment}</Text>
              </Pressable>
            </View>
          )
        } else {
                return (
                        <View>
                                <Text></Text>
                        </View>
                )
        }
    }

    return (
        <View>
        {
                showNotification()
        }
        </View>
    )
}


const Features = ({ device }) => {

    const [ deviceCharacteristics, setDeviceCharacteristics ] = React.useState([]);
    const [ settingsState, setSettingsState ] = React.useState(false);


    let characteristicsList = [];
```

```
	const discoverCharacteristics = () => {
		device.discoverAllServicesAndCharacteristics().then((desiredDevice) => {
				// Do work on device with services and characteristics
				return desiredDevice.services();
		})
		.then((services) => {
			let service = null;
			for (let i = 0; i < services.length; i++) {
				if (services[i].uuid === '12634d89-d598-4874-8e86-
7d042ee07ba7') {
					service = services[i];
				}
			}
			//return services[0].characteristics();
			return service.characteristics();
		})
		.then((characteristics) => {
			console.log(characteristics);
			//let characteristic = characteristics[1];
			for (let i = 0; i < characteristics.length; i++) {
				console.log(characteristics[i].uuid);
				characteristicsList = [...characteristicsList,
characteristics[i]];

				// characteristic = characteristics[i];
			}

			setDeviceCharacteristics(characteristicsList);


			// characteristics[0].descriptors().then((desc) => {
		//    for (let i = 0; i < desc.length; i++) {
		//        desc[i]?.read().then((val) => {
			//        if (val) {
			//          console.log(Base64.decode(val.value));
			//          console.log(val.value);
			//          console.log(val);
			//        }
			//      });
		//    }
		//   });
		})
		.catch((error) => {
			// Handle error
		});
	};

	const toggleSettingsState = () => {
		setSettingsState(!settingsState);
```

```
};

useEffect( () => {
   // Anything in here will be fired on component mount.
   console.log('Feature component mounted.');
   // const subscription = manager.onStateChange((state) => {
   //     if (state === 'PoweredOn') {
   //         scanAndConnect();
   //         subscription.remove();
   //     }
   //  }, true);
   discoverCharacteristics();
   return () => {
     console.log("Feature component Unmount.");
     //manager.destroy();
   };
  }, []);



return (
        <View>
                <View style={styles.mainBar}>
                        <Pressable onPress={ toggleSettingsState }>
                                {
                                        settingsState ?
                                                <Text
style={styles.backText}>Back</Text>
                                                :
                                                <Image
                                        style={ styles.settings }
                                        source={ require('../static/settings.png') }
                                        />
                                }
                        </Pressable>
                        {
                                deviceCharacteristics.slice(13,15).map((character, i)
=> {
                                        return (
                                                <IceNotification key={i}
iceCharacter={ character } />
                                        )
                                })
                        }
                        {
                                deviceCharacteristics.length > 0 &&
                                        <BatteryLevelIcon batteryCharacter={
deviceCharacteristics[12] }/>
                        }
```

```
                        </View>

                        {
                                !settingsState ?
                                        deviceCharacteristics.slice(0,7).map((character, i)
=> {
                                                return (
                                                        <Characteristic key={i} character={
character } />
                                                )
                                        })
                                        :
                                        <Settings characteristics={
deviceCharacteristics.slice(7,12) }/>
                        }
                </View>
        )
}

const BluetoothDevice = ({ onClick, device }) => {

        const [ deviceConnected, setDeviceConnected ] = React.useState(false);

        const connectToDevice = (desiredDevice) => {
                console.log('Device ID: ' + desiredDevice.id);
                desiredDevice.connect()
                        .then((device) => {
                                device.isConnected()
                                        .then((connected) => {
                                                console.log('Device connected: ' +
connected);
                                                setDeviceConnected(true);
                                        });

                        })

                        .catch((error) => {
                                // Handle error
                        });
        };

        const disconnectToDevice = (desiredDevice) => {
                desiredDevice.cancelConnection()
                        .then((device) => {
                                console.log('Device disconnected.');
                                setDeviceConnected(false);
                        })
                        .catch((error) => {
                                console.log('Catch operation: ' + error.message);
                                throw error;
```

```
                        });
            }

            useEffect( () => {
              // Anything in here will be fired on component mount.
              console.log('Device component mounted.');
              // const subscription = manager.onStateChange((state) => {
              //     if (state === 'PoweredOn') {
              //         scanAndConnect();
              //         subscription.remove();
              //     }
              //  }, true);
              //discoverCharacteristics();
              return () => {
                console.log("Device component Unmount.");
                //manager.destroy();
              };
            }, []);

            return (
                        //<TouchableOpacity
                        //          style={ styles.container }
                        //          onPress={ () => { connectToDevice(device) } }
                        //          >
                                    <View style={styles.container}>
                                                <Text style={styles.localName}>{ device.localName
}</Text>
                                                <View style={styles.featuresContainer}>
                                                { deviceConnected ?
                                                            <Features device={ device } />
                                                            :
                                                            <Image
                                              style={ styles.loading }
                                              source={ require('../static/connected.png') }
                                              />
                                                }
                                                </View>
                                                { deviceConnected ?
                                                            <Button onPress={ () =>
disconnectToDevice(device) } title="Disconnect" />
                                                            :
                                                            <Button onPress={ () => connectToDevice(device)
} title="Connect" />
                                                }
                                    </View>
                        //</TouchableOpacity>
            );
}

const styles = StyleSheet.create({
```

```
container: {
        flex: 1,
        alignItems: 'center',
        marginTop: 10,
        marginBottom: 10,
},
loading: {
   width: 150,
   height: 150,
   marginBottom: 20,

  },
localName: {
        fontSize: 36,
        fontWeight: 'bold',
},
featuresContainer: {
        flex: 1,
        justifyContent: 'center',
},
featureNamesText: {
        fontSize: 26,
},
dataText: {
        fontSize: 30,
        fontWeight: 'bold',
},
characteristicContainer: {
        height: 70,
        width: 300,
        flexDirection: 'row',
        justifyContent: 'space-between',
},
mainBar: {
        flexDirection: 'row',
        justifyContent: 'space-between',
},
batteryLevelIcon: {
        alignSelf: 'flex-end',
        width: 30,
        height: 50,
        margin: 10,
        marginBottom: 30,
},
settings:  {
        alignSelf: 'flex-end',
        width: 40,
        height: 40,
        margin: 10,
        marginBottom: 30,
```

```
		},
		backText: {
			alignSelf: 'flex-end',
			fontSize: 20,
			marginTop: 30,
		},
centeredView: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    marginTop: 22
  },
  modalView: {
    margin: 20,
    backgroundColor: "white",
    borderRadius: 20,
    padding: 35,
    alignItems: "center",
    shadowColor: "#000",
    shadowOffset: {
      width: 0,
      height: 2
    },
    shadowOpacity: 0.25,
    shadowRadius: 4,
    elevation: 5
  },
  button: {
    borderRadius: 20,
    padding: 10,
    elevation: 2
  },
  buttonOpen: {
    backgroundColor: "#F194FF",
  },
  buttonClose: {
    backgroundColor: "#2196F3",
  },
  textStyle: {
    color: "white",
    fontWeight: "bold",
    textAlign: "center"
  },
  modalText: {
    marginBottom: 15,
    textAlign: "center"
  }
})

export default BluetoothDevice;
```

Settings.js is settings components.

```
import React, { useEffect, useState } from 'react';
import {
	View,
	Text,
	StyleSheet,
	Switch,
	Button,
	Pressable,
} from 'react-native';
import { Base64 } from 'js-base64';

const LED8_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c150";
const LED9_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c151";
const LED10_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c152";
const LED11_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c153";
const LED12_UUID = "4116f8d2-9f66-4f58-a53d-fc7440e7c154";

const SettingsButton = ({title, isEnabled, onClick, temp}) => {



	return (
		<Pressable
			style={[styles.button, isEnabled && styles.buttonEnabled, temp
&& styles.buttonTemp]}
			onPress={ onClick }>
			<Text style={styles.textColor}>{title}</Text>
		</Pressable>
	)
}

const SettingsCharacteristic = ({ character }) => {
	const [ deviceCharacteristic, setDeviceCharacteristic ] = React.useState({ uuid: 1,
value: 'test' });
	const [ characteristicState, setCharacteristicState] = React.useState('OFF');
	const [ type, setType ] = React.useState(0);
	const [ featureName, setFeatureName ] = React.useState('');

	const setFeatureNames = (uuid) => {
		switch (uuid) {
			case LED8_UUID:
				setFeatureName('LED');
				break;
			case LED9_UUID:
				setFeatureName('LED Color');
				break;
			case LED10_UUID:
				setFeatureName('LED Effect');
```

```
				break;
			case LED11_UUID:
				setFeatureName('Lock');
				break;
			case LED12_UUID:
				setFeatureName('Temp');
				break;
			default:
				setFeatureName('Unknown');
		}
}

const setTypeByUUID = (uuid) => {
		if (uuid == LED8_UUID || uuid == LED11_UUID) {
			setType(0);
		}
		else if (uuid == LED9_UUID) {
			setType(1);
		}
		else if (uuid == LED10_UUID) {
			setType(2);
		}
		else {
			setType(3);
		}
}

const readCharacteristic = () => {
		character.read().then((characteristic) => {

			console.log('Characteristics');
			console.log(characteristic);
			console.log(characteristic.uuid);
			console.log(characteristic.value);
			//console.log(Base64.decode(characteristic.value));

			setTypeByUUID(characteristic.uuid);
			setFeatureNames(characteristic.uuid);
			setCharacteristicState(Base64.decode(characteristic.value));
			//setDeviceCharacteristic(characteristic);
		})
		.catch((error) => {
			console.log('Catch operation: ' + error.message);
			throw error;
		});
};

const sendValue = (value) => {
```

```
        character.writeWithResponse(Base64.encode(value)).then((newCharacteristic) =>
{
                        console.log("New value written.");
                        setCharacteristicState(value);
                });
        };

        useEffect( () => {
            // Anything in here will be fired on component mount.
            console.log('Characteristic component mounted.');
            // const subscription = manager.onStateChange((state) => {
            //     if (state === 'PoweredOn') {
            //         scanAndConnect();
            //         subscription.remove();
            //     }
            //  }, true);
            readCharacteristic()
            const readIntervals = setInterval(readCharacteristic, 3000);
            return () => {
              console.log("Characteristic component Unmount.");
              //manager.destroy();
              clearInterval(readIntervals);
            };
          }, []);

        return (
                <View>
                        { type === 0 &&
                                <View style={styles.characteristicContainer}>
                                        <Text
style={styles.featureText}>{featureName}</Text>
                                        <View style={styles.buttonContainer}>
                                                <SettingsButton
                                                        title="Auto"
                                                        isEnabled={ characteristicState ==
'AUTO' }
                                                        onClick={ () =>
sendValue("AUTO") }
                                                        />
                                                <SettingsButton
                                                        title="Manual"
                                                        isEnabled={ characteristicState ==
'MANUAL' }
                                                        onClick={ () =>
sendValue("MANUAL") }
                                                        />
                                        </View>
                                </View>
                        }
```

```
{ type === 1 &&
	<View style={styles.characteristicContainer}>
		<Text
style={styles.featureText}>{featureName}</Text>
		<View>
			<SettingsButton
				title="Red"
				isEnabled={ characteristicState ==
'RED' }
				onClick={ () => sendValue("RED")
}
				/>
			<SettingsButton
				title="Blue"
				isEnabled={ characteristicState ==
'BLUE' }
				onClick={ () => sendValue("BLUE")
}
				/>
			<SettingsButton
				title="White"
				isEnabled={ characteristicState ==
'WHITE' }
				onClick={ () =>
sendValue("WHITE") }
				/>
		</View>
	</View>
}
{ type === 2 &&
	<View style={styles.characteristicContainer}>
		<Text
style={styles.featureText}>{featureName}</Text>
		<View>
			<SettingsButton
				title="Solid"
				isEnabled={ characteristicState ==
'SOLID' }
				onClick={ () =>
sendValue("SOLID") }
				/>
			<SettingsButton
				title="Rainbow"
				isEnabled={ characteristicState ==
'RAINBOW' }
				onClick={ () =>
sendValue("RAINBOW") }
				/>
		</View>
	</View>
```

```
                        }
                        { type === 3 &&
                                <View style={styles.characteristicContainer}>
                                        <Text
style={styles.featureText}>{featureName}</Text>
                                        <View style={styles.buttonContainer}>
                                                <SettingsButton
                                                        title="Celsius"
                                                        isEnabled={ characteristicState ==
'CELSIUS' }
                                                        onClick={ () =>
sendValue("CELSIUS") }
                                                        temp={true}
                                                        />
                                                <SettingsButton
                                                        title="Fahrenheit"
                                                        isEnabled={ characteristicState ==
'FAHRENHEIT' }
                                                        onClick={ () =>
sendValue("FAHRENHEIT") }
                                                        temp={true}
                                                        />
                                        </View>
                                </View>
                        }
                </View>
        );
};

const Settings = ({ characteristics }) => {
        console.log("Setting characteristicssss");
        console.log(characteristics);
        return (
                <View style={styles.container}>
                        {
                                characteristics.map((characteristic, i) => {
                                        return (
                                                <SettingsCharacteristic key={i}
character={characteristic} />
                                        )
                                })
                        }

                </View>
        );
};

const styles = StyleSheet.create({
        container: {
                flex: 1,
```

```
			justifyContent: 'center',
		},
		characteristicContainer: {
			paddingVertical: 10,
			width: 300,
			flexDirection: 'row',
			justifyContent: 'space-between',
		},
		buttonContainer: {
			flexDirection: 'row',
		},
		featureText: {
			fontSize: 24,
		},
		button: {
			width: 80,
			height: 35,
			paddingVertical: 10,
			paddingHorizontal: 10,
			marginHorizontal: 10,
			marginVertical: 5,
			backgroundColor: '#81b0ff',
			borderRadius: 4,
			elevation: 3,
			textAlign: 'center',
		},
		buttonTemp: {
			width: 100,
		},
		buttonEnabled: {
			backgroundColor: '#f5dd4b',
		},
		textColor: {
			color: 'black',
			textAlign: 'center',
		},
})

export default Settings;
```

Appendix G

Lessons Learned

3D Printing

A few important points were learned about the process of 3d printing. First was making sure the printer was set to the correct settings. The filament used is ABS plastic, as it was learned that PLA can warp when outside in the heat for too long, so ABS was the suitable material. The program used to slice the models was Ultimaker Cura. Because some of the models required printing surfaces that were above the build plate, the program needed to be modified to allow printing support beams. For instance, the sliding lock that was printed needed to have a hollow case, so the program filled in the gap with support beams. However, looking at the print, the support beams made the part unusable, unless the support beams were taken out by cutting and sanding.

Another issue that can be seen is 3d printing multiple parts at once. Since the extruder moves between each part, the plastic is being dragged across the part. In the following prints, thin lines of plastic run throughout the part. Notice also that depending on the orientation of the part, the print time and material used can be reduced. For the following two prints, the orientation was upside down, seeing as the parts are hollow, if they were printed rightside up, there would be a lot of support beams to break off.

Bluetooth Technology

For this project, Bluetooth was used for the peripherals to communicate with the mobile app. When devices need to send and receive information without the use of WiFi, Bluetooth tends to be the standard set for close distance communication between devices. Bluetooth tends to be used to stream audio, or send and receive messages between devices. There are two types of Bluetooth: Bluetooth Classic and Bluetooth Low Energy.

Bluetooth Classic vs Bluetooth Low Energy (BLE)

Both Bluetooth Classic and Bluetooth Low Energy use the 2.4 GHz frequency band to send and receive information. However, there are differences between the two, thus each having different uses.

Bluetooth classic provides two-way communication with an application throughput of 2.1 Mbps [4]. It's highly effective at short distances. Bluetooth Classic is mainly used to stream audio and is commonly used for wireless speakers, headphones, and in-car entertainment systems [5]. It has a latency of 100 ms and up to 30 mA of power consumption [4].

Bluetooth Low Energy has an application throughput of 0.3 Mbps, with a limit of 20 bytes packages allowed to be send [4]. Bluetooth Low Energy is used for its communication capabilities, but is now also used for indoor positioning [5]. It has low power consumption (up to 15 mA, but can be 100x lower), with a range of 100 m and a latency of 3 ms [4].

Bluetooth Properties

Bluetooth's main advantage is being able to communicate with devices without a router or access point. To follow the Bluetooth standard, the Bluetooth device must have a profile defined so it can communicate with other devices. A GATT or general access profile (GAP) defines the records of Services, Characteristics, and Descriptors. Services are the data structure that contain Characteristics. Characteristics contain information such as its type, value, properties and permissions. The permission allowable are read, write, and notify. Read and write are self-explanatory, but notify means that when the Characteristic changes, the device connected to it will be notified of the change and read the value. Descriptors are just a general description of the Characteristic, and is optional. Each attribute must have a UUID, either a 16-bit value, or a custom 128-bit UUID [6].

Device discovery involves the defining general access profile (GAP), where the device that needs to be discovered is "advertising" its GAP, and the device scanning for the advertising device is searching and connecting to it. Advertising involves sending packets continually, so that the device scanning can detect it. The advertising device is the Bluetooth peripheral and the scanning device is the Bluetooth central [6].

Raspberry Pi

This project required a microcontroller that could utilize Bluetooth. The Raspberry Pi was selected as it would have the capabilities to control the multiple peripherals that would be needed for the Smart Cooler. Since the cooler features a GPS module, ws281x LED light strip, an RFID module, a servo, two temperature sensors, a

light sensor, a touchscreen, and switching capabilities to turn on and off the Bluetooth speakers, GPS module, and LED light strips. The cooler has a lot of features. The Raspberry Pi was chosen, as it would allow all these devices to be controlled with one microcontroller. Another benefit of why the Raspberry Pi was selected was that it has Bluetooth capabilities already on board, as well as wifi.

Setting up the Raspberry Pi involved a lot of configurations. The Raspberry Pi in this cooler uses the SPI protocol, serial port, and 1-wire communication. The different modules needed libraries to be installed to be able to communicate with the devices. For instance, the GPS module required the gpsd library to allow to use it. The LED light strips use ws281x LED lights, so a specific library for those lights were required to use the light strip. Another specification for the light strips used was that the audio on the Raspberry Pi needed to be disabled. The RFID module requires SPI. An analog-to-digital converter was used for the light sensor and battery level sensor to convert a simple voltage divider to calculate a proportional value. Luckily, the Raspberry Pi is a widely popular microcontroller, so the devices used with the Raspberry Pi were selected because they were compatible. The Raspberry Pi contains 40 pins, 26 of them GPIO pins. Some pins have special functions, such as SPI, I2C, and serial communication.

Quick Connects

For the locking mechanism, there were three devices that needed to share a ground. The servo, the RFID, and the door position sensor all shared a ground. In order to connect all three devices, a quick connect was used. The quick connect allowed the simple connection of pushing the wires into the quick connect, and then squeezing the quick connect closed to join the connections.

However, upon testing it was discovered that the ground connects from the quick connects were not connected. The door position sensor and servo was working, but not the RFID. After checking the wiring of the RFID, the conclusion was that the ground was possibly not connected. It was assumed that the wire gauge used for the quick connect was too small, therefore not providing a reliable connection. The solution to this was not to use the quick connects, but to solder all the connections together. This solved the issue.

Python Threading

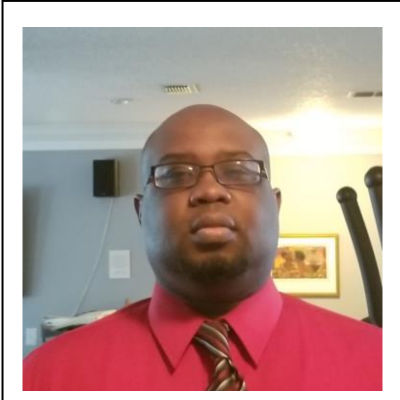
Unfortunately for the program, several loops needed to be run at once. Bluetooth needed to use a loop, the touchscreen GUI needed a loop, the LED light strip had its own loop, the

RFID required its own loop to wait for an RFID device to activate, and a separate loop was run for all the sensor data. Unfortunately, not all these interfaces could be run on the same loop, as some wait for a response, which would halt the program. To fix this, python threading was used. Threading allows the users to run code concurrently. "A thread is a separate flow of execution. This means that your program will have two things happening at once. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to" [29]. Initially there were more than five threads, however, issues started to occur where some threads weren't running, so the number was reduced.

LED Auto Feature

When the LED lights are set to auto, the Raspberry Pi reads the value of the photosensor to determine if it is dark outside. Initially there were issues as the light from the cooler was affecting the sensor value. Since the LED lights were so bright, when the lights are activated, the light reflects out of the cooler and radiates to make its surrounding area more visible, which would then cause the photosensor to tell the Raspberry Pi it was light outside. This caused a perpetual on and off cycle where the cooler is placed outside in the dark, the light would turn on, and since the photosensor was reading light values it would turn off the lights. The lights would then turn back on as it was dark outside, and the cycle would continue. To fix this, a variable was added in the program to save its state, where once the lid is open and it is dark outside, turn and keep the lights on until the lid is closed.

Biography



Clarence Scott

I am currently working full-time as an LRU Repair Technician while in pursuit of my Bachelors in Electrical and Computer Engineering Technology with a concentration in Electrical/Electronic Systems. I am expected to graduate in the spring of 2022.

Reuben Taveras

I am currently a student on his final year of school pursuing a Bachelors in Electrical and Computer Engineering Technology with a concentration in Electrical/Electronic. My interests include science, technology, web development, and art.