ELEC 490/498: Capstone Project

Final Report

Group 26 - TemperNova

Austin Greisman - [redacted]
Alex Ruffo - [redacted]
Stefan Pompilio - [redacted]
Wilson Lai - [redacted]

Supervisor: Prof. Ali Etemad

Queen's University
Electrical and Computer Engineering Department

Kingston, Ontario

April 5, 2020

Executive Summary

Canadians drink more coffee than any other beverage, including tap water. With 2/3 of Canadians drinking at least one cup a day, Canada ranks 10th in highest coffee consumption the world with approximately 7kg of coffee per capita [1]. Coffee is mostly brewed at home for breakfast, mainly by people in a rush who want to drink said coffee in a reasonable time. Currently, there are not many financially viable solutions for the consumer to fix this. Also, those solutions do not offer features that would be useful for the product, such as cooling elements, active temperature control, or a thermos locator. The team's solution to this problem is a smart temperature-controlled thermos made with inexpensive components.

The hardware of the thermos requires multiple subsystems to fulfill the project goals. As the main feature of the thermos, heating and cooling, Thermoelectric Couplers (TECs) were chosen to have a temperature differential of 50 °C to allow for heating and cooling within the optimal drinking temperature range [2]. This requirement resulted in the TEC1 12715 TECs since they allowed for over 150W of power in total [3]. To switch from heating to cooling modes on the TECs, the voltage is required to flip. Therefore, an H-Bridge MOSFET configuration was designed with two additional MOSFETs to allow for independent control of the two TECs independently on the main Printable Circuit Board (PCB). To power the TECs, a 7V converter from Texas Instruments (TI) was selected for the design as it allowed for approximately 15A of current throughput for the required temperature differential. Since our thermos is smart, a microcontroller power system was required. Therefore, a small-form-factor 5V converter from TI was designed on the main PCB. To power the entire system with approximately 100W of power, the team found that the USB-C Power Delivery (PD) protocol was the best option. Therefore, a TI USB-C PD Controller chip was integrated into our system to allow for 20V at 5A to enter the system. Finally, to allow for a closed-looped heating control system with our microcontroller, a temperature sensor was installed onto the inner wall.

The mechanical design of the thermos aims to contain all the electronic hardware components without significantly increasing the volume of the thermos. The base of the thermos was designed in Autodesk Fusion360 and is large enough to contain the PCB, Arduino, and OLED display. Screw holes are included in its design to allow attachment to the thermos body. In addition to the mechanical base, holes were drilled into the thermos body to allow for wiring for the TECs and PCB.

The software designed for the TemperNova device was broken down into two main parts: The Android companion app, and the Arduino-based microcontroller code. The Android companion app was designed around the three main software specifications: the Bluetooth LE connection, the lost device notifications and the temperature control. The Arduino code was designed around the Bluetooth LE [4] server, sensor data collection and aggregation, and the temperature control algorithm that controls the thermo-electric couplers, or TECs [3]. Combined, these pieces of software allowed for intuitive user control and management of the TemperNova thermos temperature, as well as other features.

This thermos will allow users to get their daily dose of their morning beverages quickly and be sure that they will no longer be burned when taking their first sip. The active temperature control technology mixed with the passive heat retention of the thermos provides an elegant solution to the problem that the team wished to solve. Additional features, such as location tracking and companion app control, allow the user more control over the thermos. All the features included in the thermos are provided at a much lower price than competitors offering similar products [5]. All these advantages guarantee that this product is an excellent solution for users facing the problems stated above.

Contents

Table of Figures

List of Tables

Motivation and background

The team was trying to solve the problem of actively getting morning beverages to the perfect drinking temperature and keeping them there until the user finishes their drink. Current solutions to this problem either only incorporate active heating components, or do not include any active temperature control at all. Due to the lack of proper solutions, many people burn themselves when consuming their hot beverages. When people burn their mouths and throats, they develop an increased risk of esophageal cancer [6]. This project aims to provide a cost-effective method to both heat and cool beverages so that they can reach a drinkable temperature quickly.

Design

Hardware

Hardware Overview

To achieve the desired results, it was decided that two TEC1 12715 Thermal Electric Couplers (TECs) would be used for temperature change [3]. These TECs have the capability of both heating and cooling by switching current direction. Theoretically, can heat and cool to a limit of 55°C or -5°C.

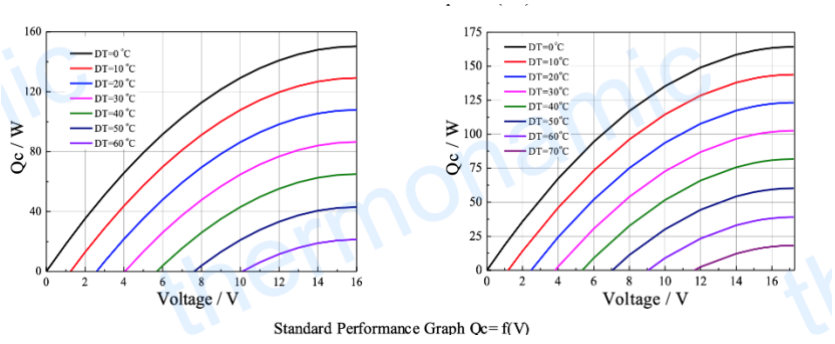

Figure 1 TEC Performance [3]

$$E = Mass * WaterCoefficient * \Delta Temp$$

$$E = 0.45kg * 4180(kJ/kg°C) * 1°\text{C} = 1881\text{J}$$

$$Time\ to\ heat\ up\ if\ we\ have\ 70W\ of\ power \rightarrow 70[\frac{J}{s}]$$

$$Time\ for\ 1°\text{C} = \frac{1881\ \text{J}}{70 J/s} \approx 27s$$

Using the above equations, the team calculated that it would take 7V at ~15A of power supplied for 27 seconds in order to raise the internal temperature of the liquid by 1°C. Thus, a 7V power converter would have to be added to the design.

Amount of Heat Required to Rise Temperature

The amount of heat needed to heat a subject from one temperature level to an other can be expressed as:

$Q = c_p\ m\ dT$ *(2)*

where

Q = amount of heat (kJ)

c_p = specific heat (kJ/kgK)

m = mass (kg)

dT = temperature difference between hot and cold side (K)

Example Heating Water

Consider the energy required to heat *1.0 kg* of water from *0 ^oC to 100 ^oC* when the specific heat of water is *4.19 kJ/kg^oC*:

Q = (4.19 kJ/kg^oC) (1.0 kg) ((100 ^oC) - (0 ^oC))

= 419 (kJ)

Figure 2 Equations for Amount of Heat Required to Raise Temperature [7]

The working voltage and current chosen for operation were 7V at 15A. These values were chosen because a 50°C temperature differential (between the hot and cold sides of the TEC) plus the estimated ambient temperature of the environment (20°C) would yield a temperature range that is ideal for hot beverages consumption [7]. Two TECs are used in the design since the efficiency of the device drastically decreases as the TECs are used. More specifically, the energy wasted by a TEC is directly proportional to the time it is on for. Noticing this, it was decided to use two TECs and activate only one at a time, alternating between the two. This allows the TEC that is off to internally equalize itself giving a higher efficiency.

Seeing that around 100W of power could potentially be needed as input to the system, the USB-C Power Delivery protocol [8] was decided as the input protocol. A pleasant side effect of this was that it allowed for the use of any USB-C charger to power our device.

Subsystems

The power delivery system consists of a USB-C PD controller chip that connects to the USB-C and feeds power to the other subsystems. The TECs will be connected to an H-bridge that alternates which TEC is powered. An Arduino with Bluetooth capabilities is needed to connect to a phone app. A temperature sensor connected to the Arduino will be used to see the temperature inside the device. An E-ink Display will be on the outside of the device, displaying the current temperature.

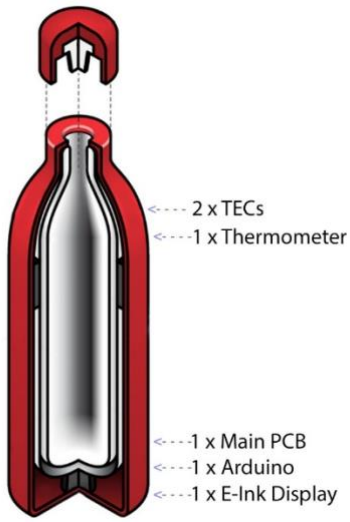

Figure 3 High Level Thermos Design

Microcontroller

The TemperNova mug hardware was designed around three main requirements; the ability to communicate via the Bluetooth LE (Low Energy) protocol to a companion app, the ability to aggregate sensor data from both the USB-C PD chip and the temperature sensor, and the ability to control the TEC states based on a simple control algorithm. While the last two requirements are straightforward, the Bluetooth LE protocol was chosen as it was designed for constantly connected devices that send small amounts of data regularly, while using significantly less power than standard Bluetooth. The major downside to the protocol however is the relative lack of supported devices; while most phones within the last 4 to 5 years support the protocol, many embedded systems and microcontrollers still do not. As such, a newer Arduino compatible microcontroller was determined to be the best option to meet these

requirements at a reasonable price. Out of all of the main Arduino families, only a few met the specs required to implement all of the above requirements; the mega family, the ESPXX based boards and their derivatives, the new zero series and the new nano series with built in Bluetooth LE chipsets [9] [10] [11] [12]. Out of these, the mega series tend to be a bit older and larger, and with the small amount of space present in the mug, these were removed from the selection process. The zero series was removed due to the significantly higher price, over twice that of the nano. There was no real difference usability wise between the ESPXX boards and the nano series, so an ESP32 board was chosen as it was sold by the retailer of choice, QKits. This board had fairly potent specs [10], with a 32 bit dual-core processor, built-in Wi-Fi b/g/n and Bluetooth 4.2 with BLE support, as well as both 5V and 3.3V pinouts.

Controlled Sensors and Components

As part of the hardware design, it was decided that adding a display on the mug itself to display the current temperature and device status would be useful for both the user. The initial hardware design used a 0.96-inch single-color OLED display, as the team already had one from a previous project [13]. As a stretch goal, a 2.13 inch 3 color E-ink display would be used, provided the Arduino could drive it at a reasonable speed [14]. For both displays, an existing display library would have to be used, as it would be extremely difficult to write one from scratch within the project timeframe.

With regards to a temperature sensor, the Dallas DS18S20 sensor [15] was chosen due to its relatively high resolution and simplicity. Accurate to ±0.5°C within -10°C to 85°C, and able to measure up to 125°C, the sensor met all the application requirements while only requiring a single data pin, unlike other sensors that required 3 or even 4 pins to work properly.

Software

Microcontroller Code

Based on these requirements, the code for the TemperNova device was designed for an Arduino microcontroller running on the ESP32 chipset platform. This modern chipset allowed for a larger codebase

as well as additional assets, making it possible to delegate more tasks to the Arduino than possible with a standard Uno or Mega microcontroller. As well, the increased CPU speed made it practical to both act as a Bluetooth LE server, and constantly poll sensors, without the threat of dropping packets due to CPU overload.

In order to drive the OLED display, the U8g2lib library was chosen as it was relatively lightweight, supported graphical characters from the Unicode character set, and allowed for arbitrary character and symbol placement within the display bounds. As for the E-ink display, there was only one choice for a library, the complex GxEPD2. This library, while supporting many useful features, used much of the 1.4 MB code space and was quite slow with respect to operations. As such, the display could not be refreshed nearly as frequently as with the OLED display, adding a design constraint for the implementation.

To create the Bluetooth LE server and to control the sending and receiving of Bluetooth packets, the *BLEDevice*, *BLEUtils*, *BLEServer* and *BLE2902* libraries were used. Combined, these libraries allowed for creating and managing a Bluetooth LE server, creating and managing services and dealing with both incoming and outgoing data. Due to the nature of the BLE protocol, the incoming and outgoing data stream support updates at both regular and irregular intervals, and automatically handle and queue messages on the Arduino side. This helps make communications relatively straightforward.

Companion App

The companion app was designed using the latest technologies and libraries for the Android platform. Built targeting Android 10, the newest version at the time, and supporting down to Android 7.0, the app is built using Kotlin, the newest language release for Android app development. It was decided to build everything using the latest technologies due to the Bluetooth LE support requirement. As a fairly new protocol, Bluetooth LE was only ratified in 2011 under the branding Bluetooth Smart and did not start appearing in most devices until around 2013 with Android 4.3 and iOS 5 [4]. Early implementations in Android were notoriously buggy, causing many developers created third party APIs and Bluetooth LE

stacks to get around these issues [16] [17] [18]. With Android 7.0 Nougat (2016), many of the previous issues were resolved, and the native Bluetooth LE stack reached relative maturity. This made it substantially easier to create the TemperNova companion app and allowed some specific features only present in newer versions of Android, such as dark mode to be implemented.

Implementation

Hardware

Microcontroller and Sensors

The hardware setup for the Arduino and related sensors was simple once the correct board schematic was found on the internet, shown below in *Figure 4*.

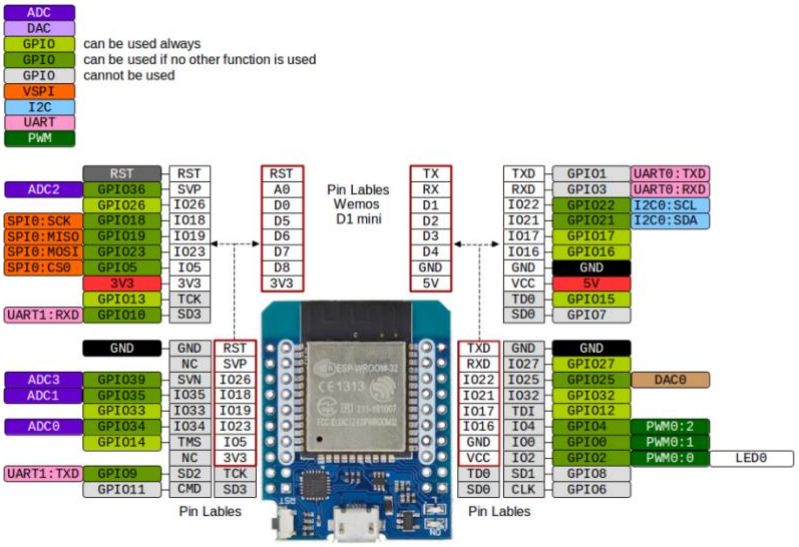

Figure 4 - Wemos Arduino ESP32 Clone Schematic [19]

Table 1 Arduino Pin Mapping

Pin	**Mode**	**Device**
21 (SDA)	Output	OLED display – serial data pin
22 (SCL)	Output	OLED display - serial clock pin
26	Input / Output	OneWire bus for temperature sensor
33	Output	Pin controlling if the TECs should be in heating mode
19	Output	Pin controlling if the TECs should be in cooling mode
35	Output	Pin controlling TEC one
18	Output	Pin controlling TEC two

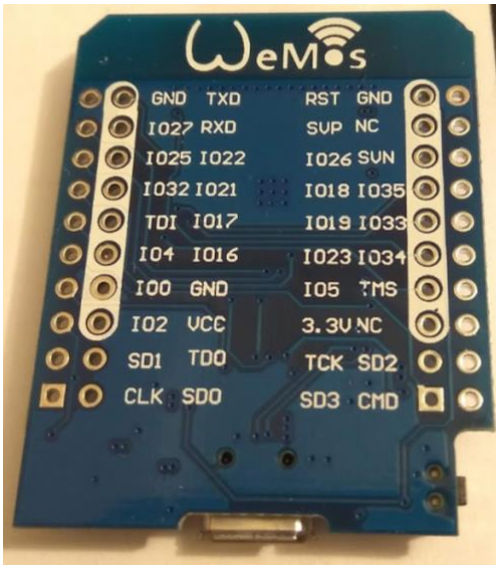

Figure 5 - Arduino Board Pinout

PCB Design

In order to connect all the subsystems together, a PCB was designed using Autodesk Eagle and professionally printed at JLCPCB in China.

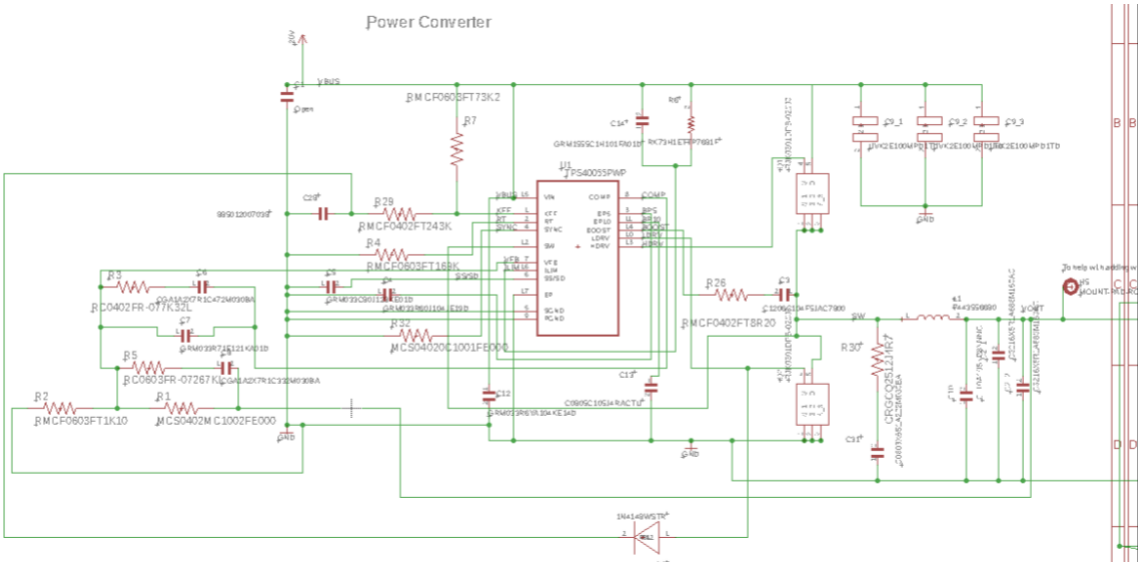

Figure 6 PCB Design

Figure 8 and Figure 9 show the two sides of the main PCB. The team manually soldered all components to the board over the course of approximately 10 hours. Figure 7 shows the efficiency of the 7V power converter. Due to the small form factor of the PCB however, placement of the 7V rail was difficult.

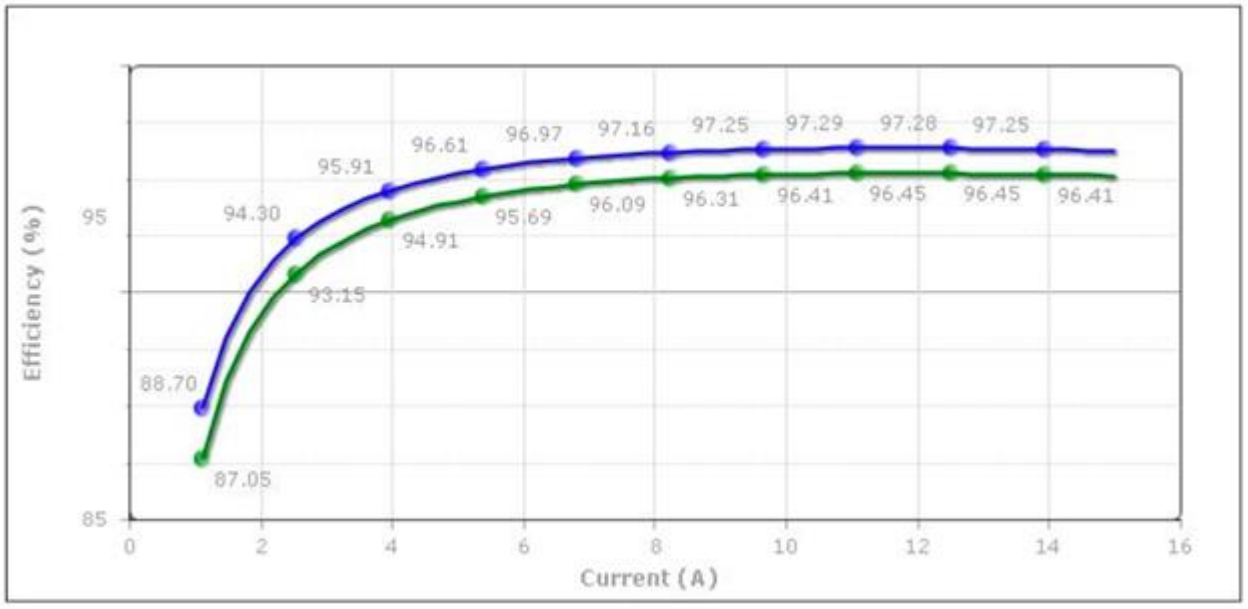

Figure 7 Efficiency of 7V Power Converter

The trace widths for the output of the 7V Converter on the PCB would be far too small and thin if designed on the PCB itself. Since our team was financially limited, a PCB with more than two layers was not acceptable. Therefore, to ensure the PCB would not catch fire if the 15A needed were to run through the traces, the final design utilized wires soldered to the bottom of the board 14AWG wire to remove this issue as seen in Figure 9.

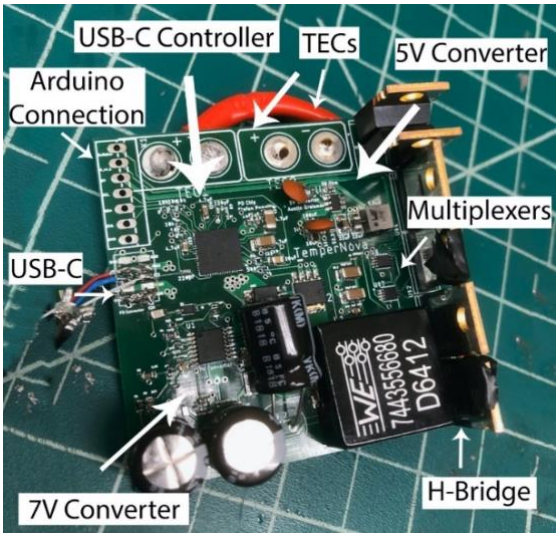

Figure 8 Physical PCB and Components

Figure 9 Back of PCB

The equation used to calculate trace width:

$$Area[mils^2] = \left(\frac{Current[A]}{k * Temp_{Rise}[°C]^b}\right)^{\frac{1}{c}} \quad where\ k = 0.048, b = 0.44, c$$

$$= 0.725\ for\ IPC - 2221$$

$$Width\ [mils] = \frac{Area[mils^2]}{Thickness[oz] * 1.378[\frac{mils}{oz}]}$$

Temperature Sensor

Initially, a digital temperature sensor from Analog electronics was chosen. Afterwards, it was discovered that the temperature sensor needed its own PCB, so a custom one was modeled in Eagle and then cut using a CNC machine, as seen in Figure 11.

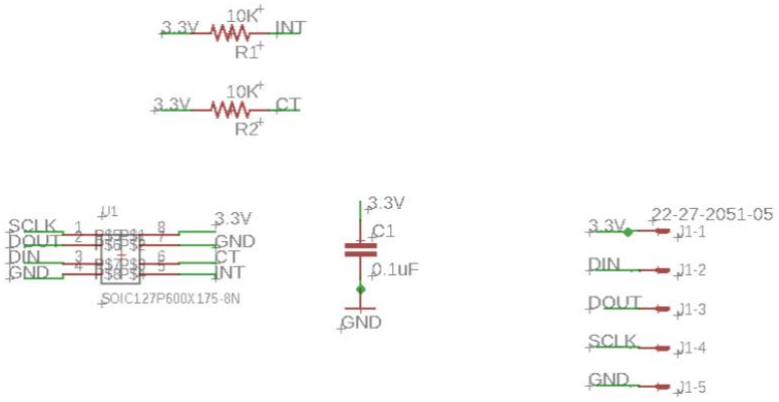

Figure 10 Temperature Sensor Design

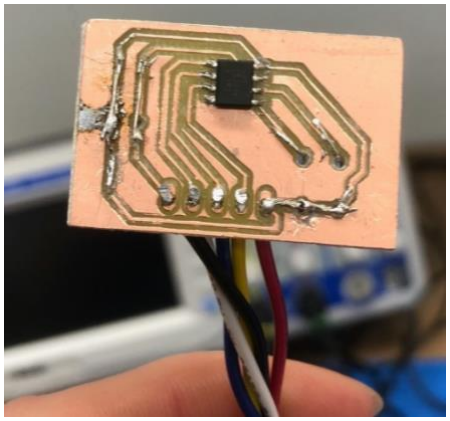

Figure 11 Onboard Analog Electronics Temperature Sensor

When testing the device, it was found that the temperature sensor was not made to operate with relatively long cables connecting the sensor to a micro controller. This resulted in the device continuously failing when trying to read and send data to and from the final PCB. It was decided to scrap this design and get a different sensor altogether. The final sensor chosen was a DS18S20 as it is natively compatibility with Arduino [15]. It also had the benefit of not needing another PCB designed for connections.

5V Bus

A 5V rail was designed to give power to the Arduino microcontroller from the USB-C input. This meant that a 5V converter had to be designed to accept 10V to 24V input. The efficiency of this converter can be seen in Figure 12. The converter would ideally never reach above 0.4A in our design, thus for our purposes, this design was satisfactory. Due to size constraints, a small form factor converter design was preferred, shown in Figure 8.

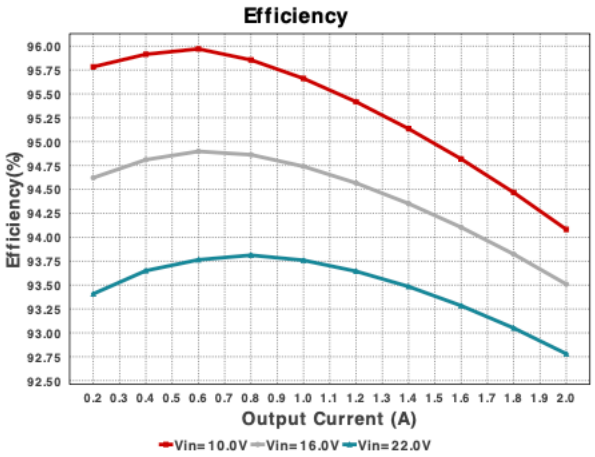

Figure 12 Efficiency of 5V Power Converter

USB-C Communication System

Using a TPS65987DDJ USB Type-C and USB PD Controller from TI, the PCB is able to communicate with the USB-C power cable and feed the necessary power to the rest of the subsystems. The recommended use outline within the chip's data sheet was mostly followed to ensure everything worked as expected. An Eagle schematic was created for this chip's connections as seen in Figure 13. An alteration to the manufacture outline was setting the chip into "Dead Battery" mode using a voltage divider. This mode allowed the chip to use the higher voltage bus (20V) within the chip as its input, rather than using pin "Vin" which required 3.3V. For its setup mode, the chip was set such that it would give a "No Wait" and maximum power configuration allowing for use of power delivery through USB-C with an input of up to 20V.

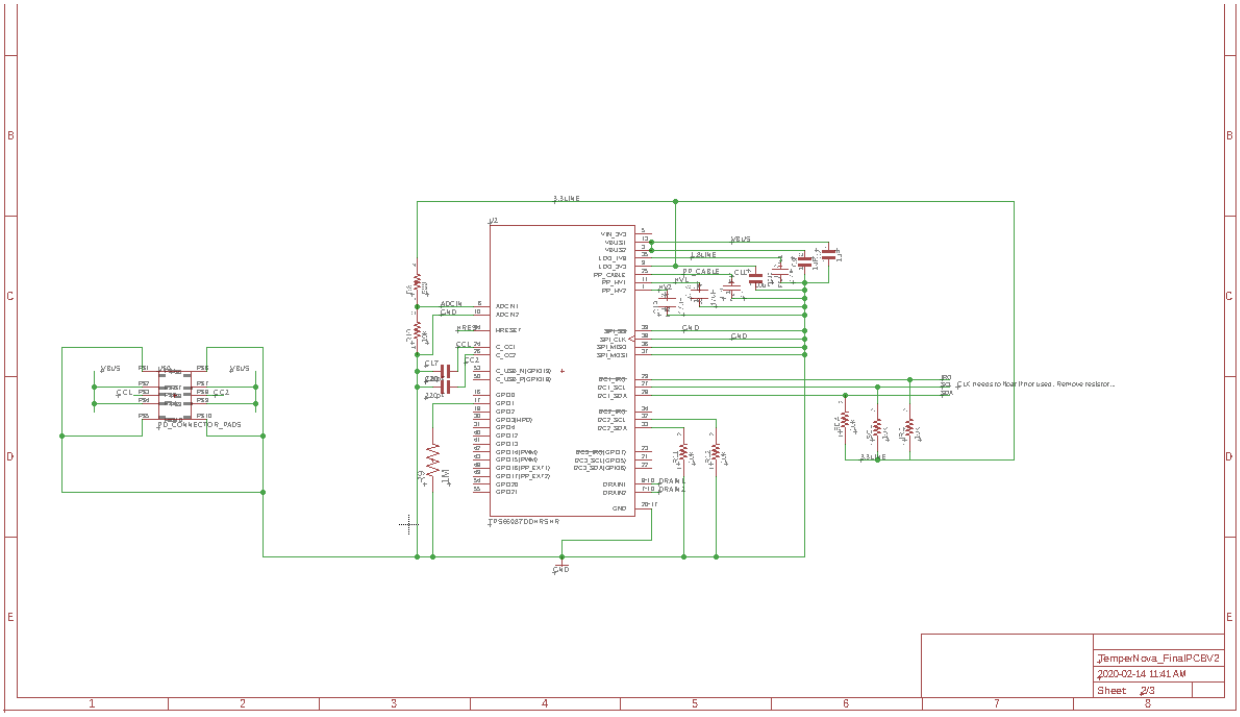

Figure 13 PCB Layout of USB-C PD Chip

Power MOSFET H-Bridge

Consisting of six IRF1324PBF High Power MOSFETs, this circuit was designed to switch each of the TECs on and off as well as flip their voltages as seen in Figure 14. Four of these MOSFETs formed an H-Bridge configuration, to allow for voltage flipping on each TEC. The last two MOSFETs were used for independent control of each TEC. The gates of these were connected to multiplexer chips that were controlled by the Arduino. Figure 15 shows a model created using LTSpice for testing the logic and connections to the multiplexers. The resistor, R, was chosen to be 0.8Ω, as at 12V, this allows for 15A to run through the system.

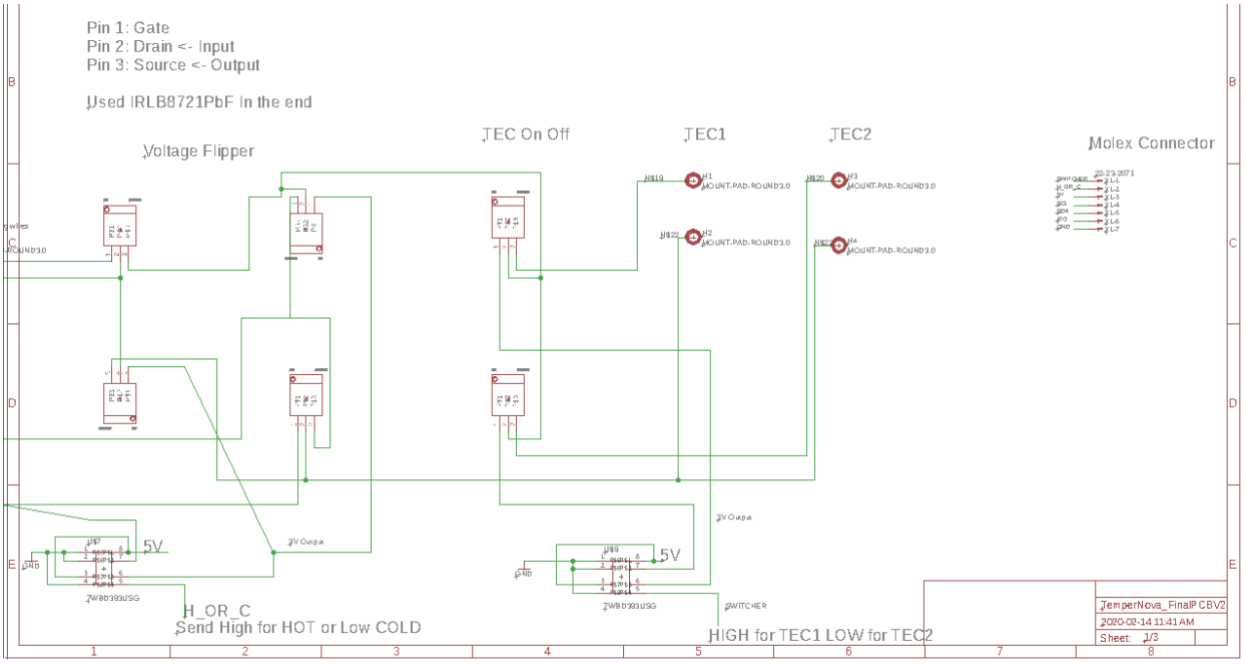

Figure 14 H-Bridge, Multiplexer Network, and ON/OFF MOSFETs

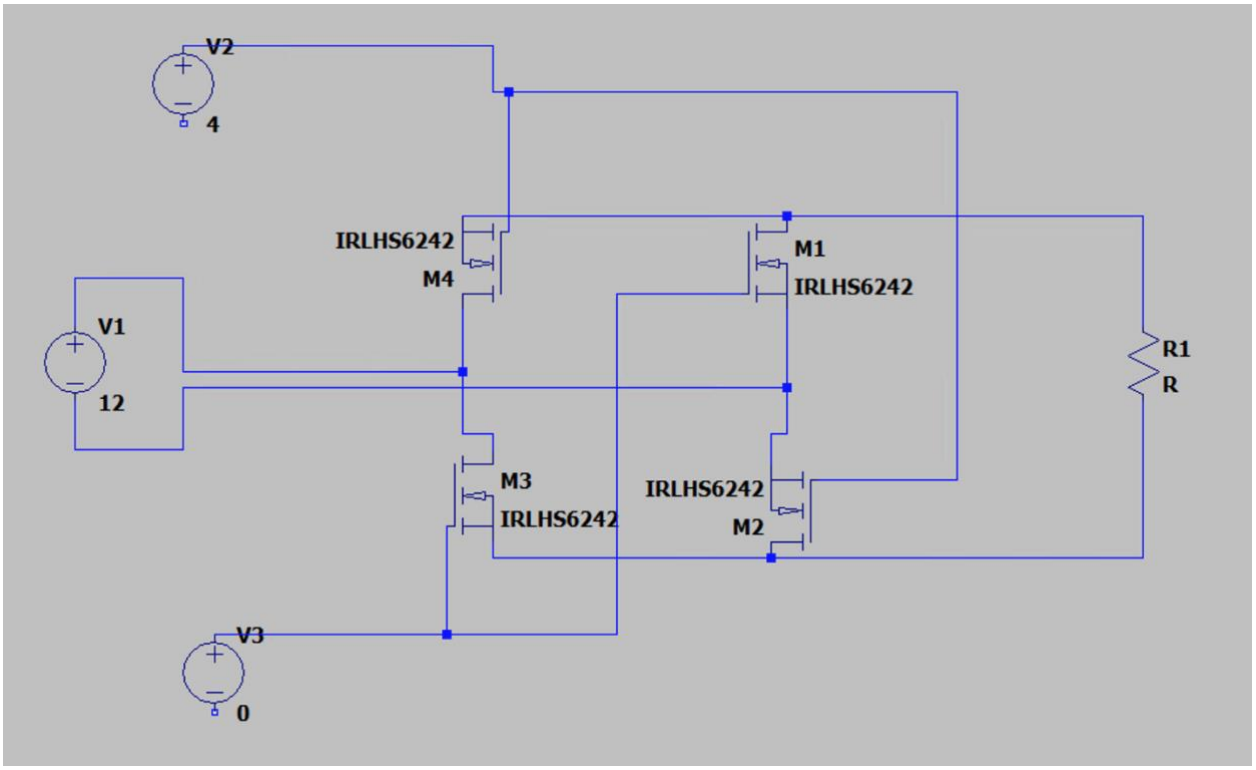

Figure 15 Simulation of H-Bridge in LTSpice

Software

Microcontroller Code

The code for the Arduino is broken into four subprograms. The main TemperNova file contains the startup code and the main program loop. The Bluetooth file contains the Bluetooth setup code and all communication functions. The Display file contains the code required to initialize the U8g2lib library and show items on the display. The Temperature file contains the code to setup the libraries and functions to both measure the temperature sensor values and control the TECs based on the current temperature value.

Main Program

The main program first calls the one-time functions for setting up Bluetooth, the temperature sensor, TECs, and initializing the display library. After the setup phase is completed, the main program loop is called as shown in Figure 16.

```
void loop() {
  int temp = getRoundedTemp();  // round the temp first to an int, might want a 1 decimal float but idk
  sendTempUpdate(temp);         // send new temp to connected BluetoothLE device, if applicable
  displayTemp(temp, true, true, true);  // int temp, bool showUnits, bool showDiff, bool showBluetoothLogo
  controlTecs();                // updates the TEC and heat / cool pins with the new state

  delay(50);  // wait a little, in order to not spam the Bluetooth connection (and because temps won't change that quickly)
}
```

Figure 16 Main Loop for Arduino

In this loop, the latest temperature reading is requested, then sent via Bluetooth LE to the companion app. If the display is currently connected, the Arduino updates the OLED and finally configures the TECs to match the desired state based on both the current, and desired temperatures. Finally, a short delay is inserted, as the temperature change is not instantaneous. This delay reduces meaningless work from occurring and prevents both the temperature sensor and Bluetooth connection from being overloaded.

Bluetooth

This file begins with the declaration of all the necessary libraries and the global variables used to keep track of the Bluetooth states and data. Figure 17 illustrates the definitions of the Bluetooth Service ID, and the Characteristic UUID.

```
#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"  // the "unique" id of the service, this can be whatever, and is random.
                                                                    // Preset values exist for common services - https://www.bluetooth.com/specifications/gatt/services/
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"  // the unique characteristic id -> this does actually have to be different
                                                                    // from all others on the same service
```

Figure 17 Bluetooth Constant Definitions

With the Bluetooth LE protocol, the method devices use to send and receive data have changed significantly from previous versions [20]. Instead of setting up what was essentially a serial communication between the two connected devices, Bluetooth LE (BLE) breaks this down into defined services and characteristics linked by a common server [21]. A service consists of a collection of characteristics that are grouped by some self-defined variable. For example, if a program were to collect humidity and temperature values, the developer might choose to cluster both characteristics into a single service representing the weather. Each characteristic can contain a single value along with zero or more descriptors that that describe the value. These human-readable descriptors can specify whatever the developer chooses, but they are commonly used for describing what that value is, a range of allowed or expected values, or the unit for the value. In the case of the TemperNova application, only a single characteristic is needed, the temperature value. During BLE transmission from the Arduino, the value in the characteristic is the last measured temperature. When receiving from the Android App, this value is interpreted as last set desired temperature. In this way, a single Bluetooth characteristic can embody all the data sent from both devices, whilst adhering to the BLE specification. As such, there is also only a single service needed, so both service and characteristic IDs are hardcoded within the program. While one could be autogenerated at startup, the specification recommends creating a single unique id to represent a service, and common services are given set IDs from the Bluetooth Special Interest Group (SIG) [22].

Hence, the value was generated once and hardcoded into the program, aligning closer to the Bluetooth spec.

Defined next, is the *setupBluetooth* function, which sets up the server along with the required service and characteristics.

```
void setupBluetooth() { // sets up the BLE connection; starts the BLE server and advertises the connection to any nearby devices.
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("TemperNova Mug");
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new BluetoothServerCallbacks());
  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
                                       CHARACTERISTIC_UUID,
                                       BLECharacteristic::PROPERTY_READ |
                                       BLECharacteristic::PROPERTY_WRITE
                                     );

  pCharacteristic->setCallbacks(new BluetoothCharacteristicCallbacks());
  pCharacteristic->addDescriptor(new BLE2902());
  pCharacteristic->setValue("69");
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);  // functions that help with iPhone connections issue
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("Characteristic defined! Starting to advertise connection...");
}
```

Figure 18 setupBluetooth *Function*

This function first creates the BLE server, passing it a new instance of the *BluetoothServerCallbacks* class shown in Figure 19.

```
class BluetoothServerCallbacks: public BLEServerCallbacks { // a collection of callbacks used in the Bluetooth server
    void onConnect(BLEServer* pServer) {    // on device connected...
      deviceConnected = true;
    };

    void onDisconnect(BLEServer* pServer) { // on device disconnected
      deviceConnected = false;
      targetTemp = 0;
    }
};
```

Figure 19 BluetoothServerCallbacks Class

BluetoothServerCallbacks contains functions that are called on “device connected” and “disconnected” events, in this case, setting a global variable to true in order to track if there is a device currently connected to the mug. This information is used to check if the Bluetooth connection logo should be shown on the display, and if the temperature updates should be sent.

Subsequently, both a new BLE service and characteristic are created with a new *CharacteristicCallbacks* class being applied to the characteristic, as shown in Figure 20.

```
class BluetoothCharacteristicCallbacks: public BLECharacteristicCallbacks { // contains the callbacks for the Bluetooth connection
    void onWrite(BLECharacteristic *pCharacteristic) {  // on information recieved from the ble characteristic from the connected device
      std::string value = pCharacteristic->getValue();  // get the raw data from the Bluetooth connection
      String valueStr = "";

      if (value.length() > 0) { // we got data!  Now we have to loop through it to make use of the data, as it is returned as a array of chars
        Serial.print("New Bluetooth value: ");
        for (int i = 0; i < value.length(); i++) {
          Serial.print(value[i]);
          valueStr+= value[i];
        }

        Serial.println();
        int temp = valueStr.toInt();  // since we know the characteristic in the app only returns numbers, we can make this assumption
        if (temp != 0) {
          targetTemp = temp;
        }
      }
    }
};
```

Figure 20 CharacteristicCallbacks Class

The *onWrite* callback is called when the BLE connection receives new data for the characteristic assigned to the callback. This function checks the received data, making sure it exists and converting it from a raw buffer array of characters into a usable string. From here, the string is converted into an integer number, and if a valid number is found, the global *targetTemp* variable is set to be the received value. A check has been implemented to ensure the temperature is not zero; this is due to an error scenario, as the buffer may sometimes have a zero constant value if a transmission or processing error occurs. As the temperature of a drink will reasonably not be set to 0°C for the scope of the prototype, this was considered a reasonable constraint. Should this product be improved upon in the future, additional checks could be substituted here in order to remove the constraint on 0°C target temperatures.

In continuation, a blank descriptor is added to the characteristic; should production occur, this should be changed to contain some basic documentation on the value and its use. However, for the purposes of the prototype, it was assumed that the optional documentation was unnecessary. A default value is then set in the characteristic in order to check that the connection is working properly on the app side. Finally, the setup function sets some remaining options for how often the server will be advertised to scanning devices, and it sets the service UUID to be the one previously defined. The server is then started, and the setup function ends.

After startup, there are a few helper functions, the first of which is the *isConnected* function. This function simply returns the connection state; true if a device is connected and false otherwise. Following this, there is the *sendTempUpdate* function, which takes a single parameter, *newTemp*, shown in Figure 21.

```
void sendTempUpdate(int newTemp) {  // send the new temp value over the BLE connection

  // check to see if device is connected...
  if (deviceConnected) {  // sanity check, obviously we should only send something if we are connected!
    char txString[8]; // make sure this is big enuff
    itoa(newTemp, txString, 10);  // call the int to string func, as the Bluetooth lib is quite picky about what data formats you can and can't send!

    pCharacteristic->setValue(txString);  // set the characteristic value to our new temp (string), this will be picked up by the respective function in the app

    pCharacteristic->notify();           // notify the connection that the data has changed!
    delay(3);                            // add a short delay to prevent spamming the connection, as the BLE connection will start to queue & drop the
                                         // packets on the app side if it can't process them fast enough!
  }
}
```

Figure 21 sendTempUpdate Function

sendTempUpdate first checks to see if there is a valid device connection, if not it aborts. If there is a valid connection, the function converts the integer temperature value into a character array buffer and sets the characteristic value to the generated buffer array. It then calls the notify function, which lets the connected device know that there is new data available. There is then a short delay, to prevent overloading the BLE connection with updates, as the custom app implementation will start to drop packets if there are too many updates to process and the message queue gets overloaded.

Finally, there is a helper function in the file called *getTargetTemp*, which simply returns the last temperature set via the app, or zero if no temperature was set.

Display

The display code is relatively simple, so only the most important function, *displayTemp* will be explained.

```
void displayTemp(int temp, bool showUnits, bool showDiff, bool showBluetoothLogo) { // displays the current temp to the OLED display.
                                                                                    // Has a few options, but it basically always called with SHowUnits as true
  if (!temp) {
    return;
  }

  u8g2.clearBuffer();         // clear the internal memory

  // Write the temp to the buffer
  u8g2.setFont(u8g2_font_logisoso22_tf);  // choose a suitable font at https://github.com/olikraus/u8g2/wiki/fntlistall

  char tempStr[4]; // make sure this is big enuff
  itoa(temp, tempStr, 10);

  if (showUnits) {  // show the "degree" symbol on screen
    u8g2.drawStr(63, 30, "\xb0"); // write something to the internal memory
  }

  u8g2.drawStr(38, 30, tempStr); // write something to the internal memory


  if (showDiff) { // show an arrow pointing in the direction of change (eg: down if the drink is hotter than the temp set in the app).
                  // Will only have an effect after the app is connected
    int targetTemp = getTargetTemp();

    u8g2.setFont(u8g2_font_open_iconic_arrow_4x_t);  // choose a suitable font at https://github.com/olikraus/u8g2/wiki/fntlistall

    if (targetTemp != 0) {
      if (targetTemp > temp)
        u8g2.drawStr(90, 31, "\x004B"); // write something to the internal memory
      else if (targetTemp < temp) {
        u8g2.drawStr(90, 31, "\x0048"); // write something to the internal memory
      }
    }
  }

  if (showBluetoothLogo && isConnected()) { // show the Bluetooth logo on screen. This is toggled via a flag in the call statement
                                            // isConnected is true when the app is connected via Bluetooth LE
    u8g2.setFont(u8g2_font_open_iconic_embedded_4x_t);  // choose a suitable font at https://github.com/olikraus/u8g2/wiki/fntlistall
    u8g2.drawStr(0, 31, "\x004A"); // write something to the internal memory
  }

  u8g2.sendBuffer();          // transfer internal memory to the display
  delay(5);                   // add a small delay
}
```

Figure 22 displayTemp Function

This function takes a few parameters, *temp*, which is the current temperature measured via the temp sensor, *showUnits*, which will show the degree symbol (°) after the current temperature if set to true, *showDiff*, which will show a directional arrow based off of the target temperature and *showBluetoothLogo*, which will show the Bluetooth logo onscreen when connected if true. The first parameter the function checks is that there was a temperature passed, if not, the function aborts. Next, the current screen buffer is cleared, and the font is set. A four-character array is generated to hold the

temperature value, and the temperature is converted from an integer value into the component number characters. Then the temperature string is written to the buffer, as well as the degree symbol if *showUnits* is set to true. Following this, if *showDiff* is set to true an arrow is drawn to the screen buffer showing the direction of temperature change. If the temperature set in the app is higher than the current temperature an up arrow is shown, and a down arrow is shown if the desired temperature is lower. Finally, if the *showBluetoothLogo* parameter is true and a BLE device is connected, the Bluetooth logo is drawn to the display as seen in Figure 23 and Figure 24.

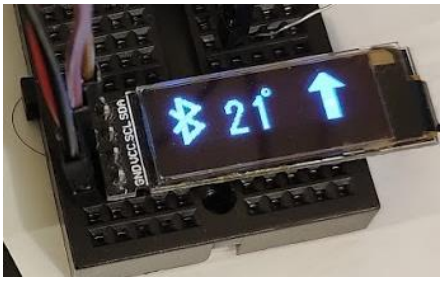

Figure 23 - OLED UI

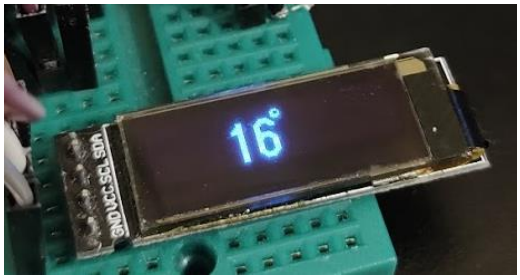

Figure 24 - OLED UI Without Bluetooth Connection

Temperature

The temperature code is more complex than the rest of the Arduino application. However, as before, the self-explanatory functions will not be explained in detail. The first task in this program is to setup the *OneWire* and *DallasTemperature* libraries. To do this, multiple constants and variables are defined.

```
#define ONE_WIRE_BUS 26 // plugged into IO 26
#define TECS_ONE_PIN 35
#define TECS_TWO_PIN 18

#define TEMP_HOT_PIN 33
#define TEMP_COLD_PIN 19

#define SWAP_COUNT 500

OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
DeviceAddress tempDeviceAddress;

int  resolution = 10;
unsigned long lastTempRequest = 0;
int  delayInMillis = 0;
float temperature = 0.0;
bool tecOneOn = false;
bool wasCooling = true;
int count = 520;
```

Figure 25 Temperature Definitions

Next, a helper function *addr2str* which takes in a *DeviceAddress* object and returns the string representation of the bus address in hex form. This is used for debugging to the console, and thus is not important to the operation of the device.

The first important function in this file is setupTempSensor in Figure 26, which initializes the *DallasTemperature* sensor object, and starts scanning for devices on the *OneWire* bus. Since there is only one *OneWire* device on the bus, the first one found is the temperature sensor. Thus, the address is set to the address of the first sensor, 0x0.

```
void setupTempSensor() {  // do the requisite setup to make sure it all works!
  sensors.begin();

  Serial.print("Locating devices...");
  Serial.print("Found ");
  Serial.print(sensors.getDeviceCount(), DEC);
  Serial.println(" devices.");

  sensors.getAddress(tempDeviceAddress, 0);

  sensors.setResolution(tempDeviceAddress, resolution);
  Serial.print("Address is: ");
  Serial.println(addr2str(tempDeviceAddress));

  sensors.setWaitForConversion(false);
  sensors.requestTemperatures();
  delayInMillis = 750 / (1 << (12 - resolution));
  lastTempRequest = millis();

  Serial.print("Delay is: ");
  Serial.print(delayInMillis);
  Serial.print(" LastTempReq: ");
  Serial.println(lastTempRequest);
}
```

Figure 26 setupTempSensor Function

As seen in Figure 26, *WaitForConversion* is then set to false, as the initial temperature reading can be done asynchronously while the remaining setup code is run. The temperature is then requested from the sensor, and the conversion delay is calculated. The sensor can only handle a set number of requests per second, as it must wait to convert the measured temperature value into a set number of bits. This delay period is directly correlated to the chosen temperature resolution, with the highest resolution taking 0.75 seconds to complete, and the lowest resolution taking just 93.75 ms, as shown in Figure 27.

Table 1. DS18B20 Conversion Times and Resolution Settings

Resolution	9 bit	10 bit	11 bit	12 bit
Conversion Time (ms)	93.75	187.5	375	750
LSB (°C)	0.5	0.25	0.125	0.0625

Figure 27 - Temperature Conversion Time [23]

If a new temperature is requested within the delay period, the sensor will drop the initial request and restart the processing time. This delay requires the application code to check how long it has been since the last call, and only request a new value if it has been longer than the minimum waiting time. Since the temperature does not change that quickly, it results in additional coding logic rather than an actual design constraint. Therefore, the *call time* is stored as a variable so it can be checked on the next temperature request.

The next interesting function is the *getTemp* function, which returns a floating-point value containing the last measured temperature. This works by first checking if the last temperature request was outside of the delay period, and if not, returning the previously measured temperature value. If it has been longer than the delay period, then the code requests a new synchronous reading from the sensor. Once the sensor is finished converting the value, it places the reading onto the bus. The value is then received by calling the *sensors.getTempCByIndex* function, which returns the temperature value in Celsius at the specified resolution. The *lastTempRequest* variable is then set to the current time in milliseconds, and the function returns the temperature.

Finally, the *controlTecs* function contains the code to control the TECs as seen in Figure 28. First, the target and current temperatures are stored in temporary variables. Next, the count is checked to see if it is an even number. This count is used for two purposes, the first being to prevent switching the TECs too quickly and reducing the number of redundant cycles within the function, as the temperature does not change often. This check causes *controlTecs* to only run once every 100 milliseconds. In addition, the count variable is used to prevent the TECs for running for more than around 24 seconds continuously. This is required as the efficiency of the TECs drops off significantly after around 25-30 seconds of continuous use.

```
void controlTecs() {    // controlls which TEC is on, and whether to heat or cool based on the current state, and which TEC was last on
  int target = getTargetTemp(); // get the target temp send from the app
  int rTemp = getRoundedTemp(); // get the rounded temp from the sensor - this is the last measured temperature reading

  if (count % 2 == 0) { // waits ~100 ms before running through the following stuff, just to make the log a bit more readable,
                        // and save some CPU processing, since the temp is not likely to change within ~100 ms :P
    if (rTemp > target) { // if we are now cooling down to reach the target temp
      if (!wasCooling) {  // if the last state was to heat, then flip the boolean to note that we are now cooling
                          // (used for making sure a TEC never goes from heating -> cooling without a slight break
        wasCooling = true;
        tecOneOn = !tecOneOn; // thus, we will flip the current TEC to the other one and reset the "timer" back to ~24 seconds
        count = SWAP_COUNT;   // reset the "timer", this is set to 520 cycles through the code, at ~60 ms / cycle
      }
      // send cooldown command
      digitalWrite(TEMP_COLD_PIN, HIGH);  // turn on "cooling" mode, and turn off heating mode
      digitalWrite(TEMP_HOT_PIN, LOW);
      Serial.print("Sending cooldown command to the TECS ");
    } else if (rTemp < target) {  // else do the opposite :)
      if (wasCooling) {
        wasCooling = false;
        tecOneOn = !tecOneOn;
        count = SWAP_COUNT;
      }

      // send heat command
      digitalWrite(TEMP_COLD_PIN, LOW);
      digitalWrite(TEMP_HOT_PIN, HIGH);
      Serial.print("Sending heat-up command to the TECS ");
    } else {  // we are at approx the right temp (after rounding, so really +- 1 degree ish), so turn off the TECs for now
        digitalWrite(TEMP_COLD_PIN, LOW);
        digitalWrite(TEMP_HOT_PIN, LOW);
        Serial.print("Sending command to turn off the TECS (it's currently at temp) ");
    }

    if (count <= 0) {   // we have reached our ~24 ish seconds, so lets swap the active TEC
      count = SWAP_COUNT;

      tecOneOn = !tecOneOn;
      Serial.println("Hit the 500 cycles, swapping which TEC is enabled");
    }

    // turn on and off to keep it at temp
    if (tecOneOn) { // turn on the respective TEC based on the current state.  This'll flip every 520 cycles (~24 ish seconds or so)
        digitalWrite(TECS_ONE_PIN, HIGH);
        digitalWrite(TECS_TWO_PIN, LOW);
        Serial.println("Using TEC 1 (TEC 2 Off)");
    } else {
        digitalWrite(TECS_TWO_PIN, HIGH);
        digitalWrite(TECS_ONE_PIN, LOW);
        Serial.println("Using TEC 2 (TEC 1 Off)");
    }
  }

  count--;
}
```

Figure 28 controlTecs Function

After the count check, the function then queries current temperature to see if it is greater than the target temperature, if so the TECs are set to cooling mode. If the previous state was heating, we also swap the TEC being used, as the TECs do not like sudden voltage changes and flipping the voltage quickly could damage the units. Should this occur, the count variable is also reset to the default of 520 cycles. Otherwise, the current TEC stays in operation and nothing happens here. Similarly, if the current temperature is

below the target temperature, the TECs are set to heating mode, and if the mode switched, the TEC is also flipped. Should the device be at the correct temperature, both TECs are turned off by setting both heating and cooling pins to off. Next, the function checks the current count value, if this is less than or equal to zero, the count is reset and the TEC in use is flipped. Finally, the function turns on the chosen TEC and turns off its companion TEC, ending the function call.

Companion App

The companion app was built in Kotlin targeting Android 10, as previously mentioned in the design specification section. The companion app has three major features; Bluetooth LE connected temperature control of the mug, disconnection notifications complete with GPS tracking, and data visualization. Each section will be broken down along with the respective helper classes and files in more complex cases.

Mug Temperature Control

The temperature of the mug can be wirelessly controlled in real-time on the app home screen, shown in Figure 29 and Figure 30.

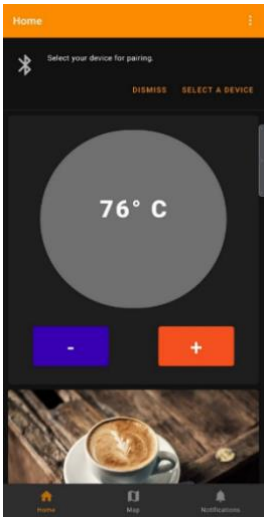

Figure 29 - App Homescreen in Disconnected State

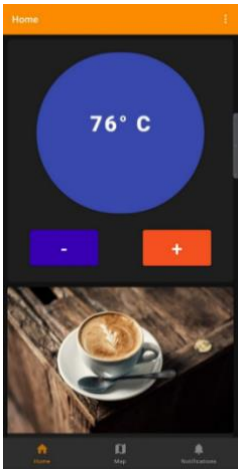

Figure 30 - App Homescreen in Connected State

The code for creating and showing this view is in *HomeFragment.kt*. However, in general the logic is straightforward, and thus only the more complex functions will be explained here.

On the initialization of the home screen, the view is created and the *checkBluetooth* function is called.

```
fun checkBluetooth(view: View) {
    print("Device has bluetooth: ")
    println((activity as MainActivity).bluetoothStatus)

    if ((activity as MainActivity).bluetoothStatus ===
Bluetooth.BluetoothStates.UNAVAILABLE) {
        displayBluetoothDisabledWarningBanner(
            view,
            getString(R.string.bluetooth_unavailable),
            getString(R.string.bluetooth_unavailable_action)
        )
    } else if ((activity as MainActivity).bluetoothStatus ===
Bluetooth.BluetoothStates.OFF) {
        displayBluetoothDisabledWarningBanner(
            view,
            getString(R.string.bluetooth_off),
            getString(R.string.bluetooth_off_action)
        )
    } else {
        if ((activity as MainActivity).bluetoothStatus ===
Bluetooth.BluetoothStates.CONNECTED) {
            (activity as MainActivity).changeDisabledState(false)
        }

        if (::banner.isInitialized) {
            banner.dismiss()
        }
    }
}
```

The *checkBluetooth* function checks to see if Bluetooth is present and, if not, displays a warning banner to the user telling them to try going into settings to turn on Bluetooth. Next, the function checks to see if Bluetooth is off, in which case a banner appears with a prompt to turn it on from within the app. Finally, the function checks to see if a Bluetooth device is already connected, in which case the banner is simply dismissed if already present. Next, the *scanForDevices* function is called, which is a wrapper around the *scanDevices* function in the *Bluetooth.kt* helper class.

Bluetooth Helper Class

The Bluetooth helper class is a full custom implementation of the Bluetooth LE specification using the native Android libraries. As such, it is quite complex and fully explaining it would take up a significant section of the report. Therefore, only the more complex or pertinent functions will be explained. For more detailed information on Bluetooth LE using the native Android libraries, see the medium blog post explaining both the protocol and required functions [24].

The Bluetooth helper class implementation begins with the declaration of multiple helper and class variables, as well as a *BluetoothProfile ServiceListener*. These are used to keep track of the internal class state, and in the case of the *ServiceListener*, provides a callback for handling connecting and disconnecting of a Bluetooth service. The first major complex function in the file, *checkBluetooth*, takes in a context and returns a *BluetoothStates* enum. A context is an Android state variable containing the global information about the application environment as well as access to global application methods, such as launching an activity. *BluetoothStates* is a custom class that returns an instance of either UNAVAILABLE, OFF, ON, CONNECTED or DISCONNECTED.

```
    fun checkBluetooth(context: Context): BluetoothStates {
        if (bluetoothAdapter == null) {
            // Device doesn't support Bluetooth
            println("Bluetooth is not detected!")
            return BluetoothStates.UNAVAILABLE
        }
```

```
        if (!bluetoothAdapter.isEnabled) {
            println("Bluetooth is turned off :(")
            return BluetoothStates.OFF
        }

        // Establish connection to the proxy.
        bluetoothAdapter?.getProfileProxy(context, profileListener,
BluetoothProfile.HEADSET)

        // ... call functions on bluetoothHeadset

        // Close proxy connection after use.
        bluetoothAdapter?.closeProfileProxy(BluetoothProfile.HEADSET,
bluetoothHeadset)

        bluetoothDeviceListAdapter = BTLEDeviceListAdapter(bluetoothDevices,
BluetoothDeviceListFragment())

        if (::bluetoothGatt.isInitialized && bluetoothGatt.device !== null) {
//            if (::bluetoothStatus.isInitialized && bluetoothStatus ===
BluetoothStates.DISCONNECTED) {
//                connectToDevice(context, bluetoothGatt.device)
//            }

            if (::bluetoothGatt.isInitialized && ::bluetoothManager.isInitialized &&
bluetoothManager.getConnectionState(bluetoothGatt.device, BluetoothProfile.GATT) ===
BluetoothProfile.STATE_DISCONNECTED) {
                connectToDevice(bluetoothGatt.device)
            } else {
                Log.d(TAG, "Has a valid connection, returning CONNECTED")
                return BluetoothStates.CONNECTED
            }
        }

        return BluetoothStates.ON
    }
```

The *checkBluetooth* function begins by performing sanity checks on the current *bluetoothAdapter* state; if it is either non-existent, meaning the device does not support Bluetooth, or if it is off, the function returns the proper state. If, however Bluetooth is both available and on, the function checks to see if there was a valid Bluetooth connection previously, if so the *connectToDevice* function is called, and an attempt to reconnect is made. The function then returns a *BluetoothStates* of ON.

Next is the *scanDevices* function. The function takes in various parameters and sets the internal class variables and states.

```
fun scanDevices(deleteStoredDevices: Boolean = false, specificDeviceToFind:
BluetoothDevice? = null, timeToScan: Long = 5000) { // from
https://medium.com/@martijn.van.welie/making-android-ble-work-part-1-a736dcd53b02
    val scanner = bluetoothAdapter!!.bluetoothLeScanner

    val scanSettings = ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
        .build()

    var scanFilters: MutableList<ScanFilter>? = null

    if (specificDeviceToFind !== null) {
        val filter = ScanFilter.Builder()
            .setDeviceAddress(specificDeviceToFind.address)
            .build()
        scanFilters = mutableListOf(filter)
    }

    if (deleteStoredDevices)
        deviceList = mutableListOf()

    if (scanner != null) {
        scanner.startScan(scanFilters, scanSettings, scanCallback) // get all devices
for now, can choose to filter by name or mac addr later.
        Handler().postDelayed({ scanner.stopScan(scanCallback) }, timeToScan)    //
scan for 5 seconds...
        Log.d("BLUETOOTH", "scan started")
    } else {
        Log.e("BLUETOOTH", "could not get scanner object")
    }
}
```

First, the scanner is initialized from the *bluetoothAdapter*, and the scan settings are set. If a specific device is requested, the scan will only return devices with a matching Bluetooth MAC address, and as these addresses are unique, a max of one device should be returned. After this configuration, if the scanner is initialized, it will start the Bluetooth scan and call the *scanCallback* function once finished execution. This callback ensures no duplicate devices are recorded and adds any new ones to the class *deviceList*.

The next interesting set of functions are the *connect* and *connectToDevice* functions. *ConnectToDevice* performs a check on all the parameters and calls the *scanDevices* function if the chosen device is not in the cache. Next, the *connect* function gets the current application context, then tries to call the *connectGatt* Bluetooth function, which calls the *bleGattCallback* function once completed.

bleGattCallback is the main function that deals with handling Bluetooth LE connection state changes. The first sub-function within this callback is the *onConnectionStateChange* function.

```
        override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int,
newState: Int) {
            if(status == GATT_SUCCESS) {
                Log.d(TAG, "New state is: $newState")

                when (newState) {
                    BluetoothProfile.STATE_CONNECTED -> {
                        // We successfully connected, proceed with service discovery

                        val bondState: Int = gatt.device.bondState
                        // Take action depending on the bond state
                        // Take action depending on the bond state
                        if (bondState == BOND_NONE || bondState == BOND_BONDED) { //
Connected to device, now proceed to discover it's services but delay a bit if needed
                            var delayWhenBonded = 0
                            if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.N) {
                                delayWhenBonded = 1000
                            }
                            val delay =
                                if (bondState == BOND_BONDED) delayWhenBonded else 0
                            val discoverServicesRunnable = Runnable {
                                Log.d(
                                    TAG,
                                    java.lang.String.format(
                                        Locale.ENGLISH,
                                        "discovering services of '%s' with delay of
%d ms",

                                        gatt.device.name,
                                        delay
                                    )
                                )
                                val result = gatt.discoverServices()
                                if (!result) {
                                    Log.e(TAG, "discoverServices failed to start")
                                }
//                              discoverServicesRunnable = null
                            }
                            bleHandler.postDelayed(discoverServicesRunnable,
delay.toLong())
                        } else if (bondState == BOND_BONDING) { // Bonding process in
progress, let it complete
                            Log.i(TAG, "waiting for bonding to complete")
                        }

                        gatt.discoverServices()
                    }
                    BluetoothProfile.STATE_DISCONNECTED -> {
                        // We successfully disconnected on our own request
                        gatt.close()
                    }
```

```
                    else -> {
                        // We're CONNECTING or DISCONNECTING, ignore for now
                        Log.d(TAG, "New state is: $newState")
                        super.onConnectionStateChange(gatt, status, newState)
                    }
                }
            } else {
                // An error happened...figure out what happened!
                Log.d(TAG, "New state is: $newState")

                if (newState === BluetoothProfile.STATE_DISCONNECTED || newState ===
BluetoothProfile.STATE_DISCONNECTING) {
                    // handle location saving here, as we are disconnecting from the
device...

                    val context = MainActivity.CoreHelper.contextGetter?.invoke()

                    if (context !== null) {
                        (context as MainActivity).bluetoothStatus =
BluetoothStates.DISCONNECTED
                        (context as
MainActivity).locationHelper.addLastKnownLocationToList(context as MainActivity)
                        (context as MainActivity).runOnUiThread {
                            (context as MainActivity).dismissBanner()
                            (context as
MainActivity).displayBluetoothDisconnectedBanner(
                                (context as MainActivity).findViewById(
                                    R.id.nav_host_fragment
                                ), "${gatt.device.name} Disconnected"
                            )
                            (context as MainActivity).changeDisabledState(true)
                            (context as MainActivity).updateTemp((context as
MainActivity).findViewById(
                                R.id.nav_host_fragment
                            ))
                        }
                    }
                }
                gatt.close()
            }
        }
```

The *onConnectionStateChange* function keeps the class state variable updated as the connection state changes, and updates various UI segments based on the new state.

Next are the *onCharacteristicRead*, *onCharacteristicWrite* and *onCharacteristicChanged* functions. These functions deal with the incoming and outgoing data within a BLE characteristic, handling the incoming temperature values and writing the outgoing values to the characteristic properly. The

onCharacteristicChanged function is called every time the value within the BLE characteristic is changed after the initial value comes in and is essentially the same function as *onCharacteristicWrite*.

Next, there is the *onServicesDiscoverd* function, which collects all the services offered by the BLE device and finds the characteristics for each one present within the service, calling the *onCharacterisitcChanged* function for each.

Finally, there is the *onDescriptorWrite* function, which sets up a BLE descriptor value for the service, turns on or off notifications for outgoing values and sends the updated value to the Arduino if notifications are set to on.

```
override fun onDescriptorWrite(
    gatt: BluetoothGatt,
    descriptor: BluetoothGattDescriptor,
    status: Int
) {
    // Do some checks first
    val parentCharacteristic: BluetoothGattCharacteristic = descriptor.characteristic
    if (status != GATT_SUCCESS) {
        Log.e(TAG, String.format("ERROR: Write descriptor failed -> characteristic:
%s", parentCharacteristic.uuid))
    }

    // Check if this was the Client Configuration Descriptor
    if (descriptor.uuid == UUID.fromString(CCC_DESCRIPTOR_UUID)) {
        if (status == GATT_SUCCESS) {
            // Check if we were turning notify on or off
            val value: ByteArray = descriptor.value

            if (value != null) {
                if (value[0] != Byte.MIN_VALUE) {
                    // Notify set to on, add it to the set of notifying
characteristics
                    notifyingCharacteristics.add(parentCharacteristic.uuid)

                    sendDesiredTemp() // send temp back -> this way the Arduino knows
what the initial setting is...
                }
            } else {
                // Notify was turned off, so remove it from the set of notifying
characteristics
                notifyingCharacteristics.remove(parentCharacteristic.uuid)
            }
        }
```

```
        // This was a setNotify operation
        Log.d(TAG, "onDescriptorWrite called - ${descriptor.characteristic} -
${descriptor.value.toString(
            Charset.defaultCharset())}")

    } else {
        // This was a normal descriptor write....
        super.onDescriptorWrite(gatt, descriptor, status)
    }

    completedCommand()
}
```

The final interesting function in the Bluetooth helper class is the *handleTempCharacteristic* function below, which handles the incoming raw temperature data string from the Arduino.

```
fun handleTempCharacteristic(gatt: BluetoothGatt, characteristic:
BluetoothGattCharacteristic, setNotifications: Boolean = false) {
    val value = characteristic!!.value.toString(Charset.defaultCharset())

    if (value.isNotEmpty() && value.toIntOrNull() !== null) {
        if (!::notifyingGattCharacteristic.isInitialized) {
            notifyingGattCharacteristic = characteristic
        }

        val temp = value.toInt()

        val context = MainActivity.CoreHelper.contextGetter?.invoke()

        if (context !== null && temp > -100) {
            (context as MainActivity).currTemp = temp
            (context as MainActivity).changeDisabledState(false)
            context.temperatureClass.updateAvgTemp(temp)
            (context as MainActivity).runOnUiThread {
                (context as MainActivity).updateTemp(
                    (context as MainActivity).findViewById(
                        R.id.nav_host_fragment
                    )
                )
            }
        }

        if (setNotifications) {
            setNotify(characteristic, true)
        }
    }
}
```

This function performs checks on the incoming data, converts it to a standardized string, then checks to see if the value is a valid integer. If it is, checks are made to ensure a valid number is present, and not an error code, represented by large negative values. If the new value passes all the checks, the temperature

display on the home screen is updated with the new temperature value and, if disabled, is set to be re-enabled.

Mechanical

A cylindrical base as seen in Figure 31 was designed in Autodesk Fusion360 holds the PCB, Arduino, and OLED display. Holes for screws were drilled into the 3D printed base to allow for attachment to the thermos. A hole is drilled into the base of the thermos to allow for wires to connect the TECs to the PCB. This base attaches to the outer shell of the thermos which has been cut horizontally to allow for attachment of the necessary components as seen in Figure 32.

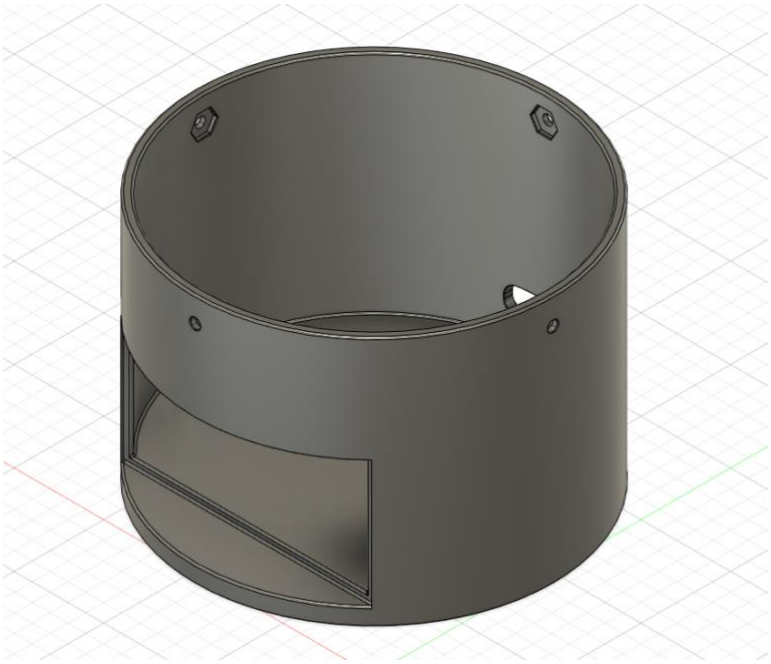

Figure 31 Mechanical Base

Figure 32 Final Inner Design of Thermos

Figure 32 illustrates the final configuration and set up of the TECs on the thermos. Forced deformation with a hammer was required to allow for both TECs to fit onto the thermos due to their rectangular and flat design. Six copper heatsinks were placed onto both TECs to dissipate the current temperature on the outer side of the TEC to the outer thermos wall through conduction. This allowed for the temperature differential of the TECs to stay high, increasing the efficiency.

Bill of Materials

Table 2 Bill of Materials

Description	Part Number	Quantity	Unit Price	Extended Price
CAP CER 0.1UF 50V X7R 0402	490-13342-1-ND	10	$ 0.06	$ 0.60
RES SMD 267 OHM 1% 1/10W 0603	311-267HRCT-ND	5	$ 0.14	$ 0.70
RASPBERRY PI COPPER HEATSINK	1738-1006-ND	12	$ 1.38	$ 16.56
CAP CER 220PF 10V C0G/NP0 0402	732-7433-1-ND	1	$ 0.13	$ 0.13
RES SMD 12K OHM 0.25% 1/16W 0402	P12KCQCT-ND	1	$ 0.80	$ 0.80

USB Type-C™ and USB PD controller	TPS65987DDHRSHR	2	$ 4.22	$ 8.44
10µF Capacitor	1276-6456-1-ND	2	$ 0.13	$ 0.26
4.7µF Capacitor	1276-1907-1-ND	4	$ 0.10	$ 0.40
1µF Capacitor	1276-1010-1-ND	2	$ 0.10	$ 0.20
47µF Capacitor	1276-6771-1-ND	1	$ 0.71	$ 0.71
220pF Cap	732-7433-1-ND	2	$ 0.10	$ 0.20
1MΩ Resistor	MCS0402-1.00M-CFCT-ND	1	$ 0.30	$ 0.30
500Ω Resistor	541-1905-1-ND	1	$ 1.13	$ 1.13
10kΩ Resistor	311-10KNCT-ND	7	$ 0.10	$ 0.70
CAP CER 0.1UF 35V X5R 0201	490-10430-1-ND	1	$ 0.13	$ 0.13
CAP CER 100PF 50V C0G/NP0 0402	490-8180-1-ND	1	$ 0.13	$ 0.13
CAP CER 68UF 10V X5R 1206	445-14673-1-ND	2	$ 2.06	$ 4.12
CAP CER 0.1UF 50V U2J 1206	399-16900-1-ND	1	$ 3.49	$ 3.49
CAP CER 2200PF 10V X6S 0201	445-13705-1-ND	1	$ 0.14	$ 0.14
CAP CER 4700PF 16V X7R 0201	445-12078-1-ND	1	$ 0.16	$ 0.16
CAP CER 3300PF 16V X7R 0201	445-12074-1-ND	1	$ 0.16	$ 0.16
CAP ALUM 10UF 20% 250V RADIAL	493-12635-1-ND	3	$ 0.83	$ 2.49
DIODE GEN PURP 75V 300MA SOD323	1655-1359-1-ND	1	$ 0.17	$ 0.17
FIXED IND 6.8UH 18.5A 4.1 MOHM	732-2174-1-ND	1	$ 8.15	$ 8.15
MOSFET N-CH 30V 60A 5-LFPAK	RJK0301DPB-02#J0CT-ND	2	$ 3.02	$ 6.04
RES SMD 10K OHM 1% 1/10W 0402	MCS0402-10.0K-MFCT-ND	1	$ 0.41	$ 0.41
RES 1.1K OHM 1% 1/10W 0603	RMCF0603FT1K10CT-ND	1	$ 0.15	$ 0.15
RES 8.2 OHM 1% 1/16W 0402	RMCF0402FT8R20CT-ND	1	$ 0.15	$ 0.15
RES 243K OHM 1% 1/16W 0402	RMCF0402FT243KCT-ND	1	$ 0.15	$ 0.15
RES SMD 7.32K OHM 1% 1/16W 0402	YAG3236CT-ND	1	$ 0.14	$ 0.14
CRGCQ 2512 4R7 5%	A130292CT-ND	1	$ 0.61	$ 0.61

RES SMD 1K OHM 1% 1/10W 0402	MCS0402-1.00K-CFCT-ND	1	$ 0.41	$ 0.41
RES 169K OHM 1% 1/10W 0603	RMCF0603FT169KCT-ND	1	$ 0.15	$ 0.15
RES SMD 267K OHM 1% 1/10W 0603	311-267KHRCT-ND	1	$ 0.14	$ 0.14
RES 7.68K OHM 1% 1/10W 0402	2019-RK73H1ETTP7681FCT-ND	1	$ 0.14	$ 0.14
RES 73.2K OHM 1% 1/10W 0603	RMCF0603FT73K2CT-ND	1	$ 0.15	$ 0.15
IC REG CTRLR BUCK 16HTSSOP	296-22445-2-ND	1	$ 0.50	$ 0.50
CAP CER 0.1UF 6.3V X5R 0201	490-16364-1-ND	1	$ 0.13	$ 0.13
CAP CER 470PF 25V C0G/NP0 0805	732-7837-1-ND	1	$ 0.13	$ 0.13
CAP CER 1UF 10V X5R 0603	1276-1182-2-ND	1	$ 0.02	$ 0.02
CAP CER 1UF 16V X7R 0805	399-8001-2-ND	1	$ 0.02	$ 0.02
CAP CER 0.012UF 6.3V X6S 0201	490-11351-1-ND	1	$ 0.13	$ 0.13
CAP CER 120PF 25V X7R 0201	490-3173-1-ND	1	$ 0.13	$ 0.13
SENSOR DIGITAL -55C-150C 8SOIC	ADT7310TRZ-REEL7TR-ND	1	$ 5.00	$ 5.00
CAP CER 10UF 10V X5R 0805	1276-6456-1-ND	2	$ 0.17	$ 0.34
CAP CER 4.7UF 6.3V X5R 0603	1276-1907-1-ND	4	$ 0.13	$ 0.52
CAP CER 1UF 6.3V X5R 0402	1276-1010-1-ND	2	$ 0.13	$ 0.26
CAP CER 47UF 6.3V X6S 1206	1276-6771-1-ND	1	$ 0.92	$ 0.92
CAP CER 220PF 10V C0G/NP0 0402	732-7433-1-ND	1	$ 0.13	$ 0.13
RES SMD 1M OHM 1% 1/10W 0402	MCS0402-1.00M-CFCT-ND	1	$ 0.41	$ 0.41
RES SMD 500 OHM 0.1% 1/20W 0402	541-1905-1-ND	1	$ 1.55	$ 1.55
RES SMD 10K OHM 5% 1/20W 0201	311-10KNCT-ND	7	$ 0.14	$ 0.98
Thermoelectric Cooler Peltier (TEC1-12715)	TEC1 12715	4	$ 4.26	$ 17.04
Wemos MINI D1 ESP32	WEMOSMINID1	1	$ 20.58	$ 20.58
Wide Input (8V-40V) Buck Converter	TPS40055PWP	2	$ 4.58	$ 9.16

CAP CER 0.1UF 35V X5R 0201	490-10430-1-ND	1	0.13	$	0.13
CAP CER 100PF 50V C0G/NP0 0402	490-8180-1-ND	1	0.13	$	0.13
CAP CER 68UF 10V X5R 1206	445-14673-1-ND	2	2.06	$	4.12
CAP CER 0.1UF 50V U2J 1206	399-16900-1-ND	1	3.49	$	3.49
CAP CER 2200PF 10V X6S 0201	445-13705-1-ND	1	0.14	$	0.14
CAP CER 4700PF 16V X7R 0201	445-12078-1-ND	1	0.16	$	0.16
CAP CER 3300PF 16V X7R 0201	445-12074-1-ND	1	0.16	$	0.16
CAP ALUM 10UF 20% 250V RADIAL	493-12635-1-ND	3	0.83	$	2.49
DIODE GEN PURP 75V 300MA SOD323	1655-1359-1-ND	1	0.17	$	0.17
FIXED IND 6.8UH 18.5A 4.1 MOHM	732-2174-1-ND	1	8.15	$	8.15
MOSFET N-CH 30V 60A 5-LFPAK	RJK0301DPB-02#J0CT-ND	2	3.02	$	6.04
FIXED IND 5.6UH 3.6A 51 MOHM SMD	732-11203-1-ND	1	2.39	$	2.39
PWR MGMT SPECIALIZED REGULATOR	296-53568-1-ND	1	1.76	$	1.76
RES SMD 52.3K OHM 1% 1/10W 0603	541-52.3KHCT-ND	1	0.15	$	0.15
RES SMD 10K OHM 1% 1/10W 0603	541-10.0KHCT-ND	1	0.15	$	0.15
CAP CER 0.1UF 50V X7R 0402	445-6899-1-ND	2	0.21	$	0.42
CAP CER 22UF 10V X7S 0805	445-14560-1-ND	2	1.25	$	2.50
RES SMD 30 OHM 1% 1/10W 0603	311-30.0HRCT-ND	1	0.14	$	0.14
CAP CER 10UF 25V X5R 0603	445-9015-1-ND	2	0.98	$	1.96
CAP CER 47UF 6.3V X6S 1206	1276-6771-1-ND	1	0.91	$	0.91
IC BUS SWITCH 1X1 US8	7WBD383USGOSCT-ND	2	0.79	$	1.58
			TOTAL	$	154.00
				Not including Shipping or tax	

Paid for personally by team	Part Number	Quantity	Unit Price	Extended Price
Power MOSFETs	irf1324pbf	6	$ 1.50	$ 9.00
Thermos	N/A	1	$ 20.00	$ 20.00
3D Printing Filament	N/A	4	$ 0.03	$ 0.12
Screws and Nuts	N/A	8	$ 0.05	$ 0.40
OLED Display	N/A	1	$ 7.00	$ 7.00
E-Ink Display	N/A	1	$ 10.00	$ 10.00
Wires	N/A	10	$ 0.10	$ 1.00
		TOTAL with Team		$ 201.52
				Not including Shipping, tax estimated

Hardware Testing and Evaluation

When testing sensitive electronics, it is important to always use equipment that protects against electrostatic discharge (ESD) as to not damage components. The main PCB is an example where testing and construction requires ESD safe equipment. Therefore, it was built using ESD protective equipment including an ESD grounded mat, ESD grounded arm strap, ESD safe tweezers, and multiple ESD grounded soldering irons. Each individual hardware subsystem was tested individually using simulations and physical power load testing if possible. Comparisons between expected values and real measurements were discussed in previous sections and reasons for the discrepancies were also explained.

However, the USB-C Chip implementation was bypassed in the final design with a DC Power supply. This was because it would not properly negotiate 20V with maximum power draw from any USB-C Charger connected. After some extended research, it was found that firmware needed to be flashed onto the chip to allow for this power configuration. This information was not listed on the TI site, nor the datasheet for the chip. The information was instead found on a site for hobbyists and technical enthusiasts [25]. Luckily, the team had the forethought to connect an I^2C bus to the USB-C PD chip in case such a problem arose. However, after a week of communication back and forth with TI, it was found the device needed to flash

the firmware using I^2C was over $400. This was the only tool that had been tested by TI while flashing the firmware, and as such, it was substantially out of budget to continue using this chip.

As well, an unforeseen circumstance of using high power MOSFETs was that the voltage bias needed to open the channels fully was more than the initially expected 5V. With the 5V multiplexer used in our system, all MOSFET channels were not opened wide enough to allow for 12V to pass through. This was evident when debugging the system and noticing a 3.4V output at the TECs instead of the expected 12V. After more research, it was found a common method for biasing a power MOSFET was to use another, lower powered MOSFET to bias the bigger power MOSFET. However, this discovery was made near the end of the design of the main PCB, so this fix was not implemented. This key factor was missed in our simulations seen in Figure 15 since the actual MOSFET used, IRF1324PBF, was not available as a model in LTSpice. If the project was done again, we would ensure that either we create our own model of the MOSFETs before construction or find a MOSFET in LTSpice that closely resembles our own.

Stakeholder Needs

In order to consider this product a success, it needed to be affordable, be able to cool a beverage down to a drinkable temperature quickly, be easy to locate using the companion app, and be able to contain all the components within the thermos allowing it to be easily portable.

The overall cost of the thermos that was aimed for was around $60 CAD, a much lower cost than that of the closest competitor, Ember [5]. The most important aspect of the project was being able to change the temperature of the beverage quickly. By incorporating two TECs, the temperature of the beverage was able to reach desired temperatures much faster than if only one was used. Being able to locate the thermos through the app was done exactly as specified. If the thermos is within Bluetooth range of the user's smartphone, then the location of the thermos is logged at certain intervals of time. Should the connection be lost, a notification is sent to the user with the last known location of the thermos. This method means that the product does not actively track the user's location, only that of the thermos,

allowing for the user's privacy to still be maintained. Lastly, the thermos was meant to be portable while fulfilling its function. While the team was unable to make the active heating and cooling elements of the project completely portable, the user would still be able to bring the thermos from location to location with ease. The only difference from the original idea would be that the thermos would need to be connected to a USB-C cable to actively heat or cool the beverage. Once the beverage reached a desired temperature (set by the user via the companion app) the natural insulation of the thermos would passively keep the beverage at that temperature for hours on end, maintaining the product's portability.

Compliance with Specifications

Software Specifications

Table 3 Software Specs Table

	Specification	**Specification met? (Yes/No)**
1	**Functional requirements**	
1.1	BLE communication between companion app and Arduino	YES
1.2	Real-time temperature updates in-app	YES
1.3	BLE disconnection notifications in-app	YES
1.4	Real-time temperature control in-app	YES
1.5	Temperature sensing on the Arduino to within 1°C accuracy	YES
1.6	TEC control on the Arduino based on current and desired temperatures	YES
2	**Interface requirements**	
2.1	Dynamic app interface that changes based on BLE connection state	YES
2.2	Ability to view number of drink refills	YES
2.3	Ability to view temperature changes over time	YES
2.4	Ability to view past locations of mug on disconnection	YES
3	**Performance requirements**	
3.1	App works without crashing on multiple devices and OS versions	YES
3.2	Bluetooth message queue processed in real-time	YES

Hardware Specifications

Table 4 Hardware Specs Table

	Specification	**Target value**	**Tolerance**	**Achieved Value**
1	*Peltier Current*	*15 A*	*+/- 1 A*	*1 A*
2	*Peltier Voltage*	*7 V*	*+/- 1 V*	*3.54 V*

3	*Control Device Current*	*0.2 A*	*+/- 0.5 A*	*0.2 A*
4	*Control Device Voltage*	*5 V*	*+/- 1V*	*5.01 V*
5	*Temperature Sensor*	*N/A*	*+/- 5° C*	*+/- 2° C*
6	*7V Converter*	*7 V / 15A*	*+/- 0.5V / 2A*	*7.01V / 14A*
7	*5V Converter*	*5V / 0.5A*	*+/- 0.1V / 0.2A*	*5.03V / 2A*
8	*H-Bridge*	*7V in 7V out*	*+/- 1V*	*7.01V in 3.54V out*

Conclusions and Recommendations

Technical

After completing this project, many technical lessons were learned in every area of its design and implementation. On the software side, the Android app development was slowed due to the variety of Android versions and the lack of a standardized native Bluetooth implementation, thus, one needed to be designed from scratch. In the future, it would be advisable to have someone on the team who has experience in Android app development while also having other members of the team take on some of the programming load. This is so that one member is not responsible for most of the software aspect of the project. On the hardware side of the project, the team did not initially know that there was an extreme lack of documentation for most of the parts used in the project. Knowing this, it would be advisable to research parts that have proper documentation for use and implementation before ordering them in bulk. On the mechanical side of the project, the biggest drawback was that the team was comprised entirely of electrical and computer engineers with little experience in CAD design. In the future, the team would work with a mechanical engineer who could design more complicated mechanical models that would work better for the purposes of the project.

Commercialization

This product can be commercialized and brought to the market if the thermoelectric effect, used by the TECs, can be made more efficient and have larger temperature differentials. If the product were to make it to market, it would allow users to go through their morning routines much faster and provide them with the knowledge that they will no longer burn themselves when trying to enjoy their morning beverage [6]. It is possible that rates of throat cancer will decrease as fewer people will be burning themselves. A

potential adverse societal effect may include users who are already cost sensitive spending more money on a thermos that would otherwise have cost $20 or less.

Work Breakdown

Name	Overall effort expended (%)
Austin Greisman	100
Alex Ruffo	100
Stefan Pompilio	100
Wilson Lai	100

References

[1] CoffeeBI Editorial Team, "The Canadian coffee consumption 2019," coffeebicoffeebi, 18 Feb 2019. [Online]. Available: https://coffeebi.com/2019/02/18/the-canadian-coffee-consumption-2019/.

[2] H. L. M. O'Mahony, "At What Temperatures Do Consumers Like to Drink Coffee?: Mixing Methods," 20 Jul 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2621.2002.tb08814.x.

[3] thermonamic, "Specification of Thermoelectric Module," [Online]. Available: http://www.thermonamic.com/TEC1-12715-English.PDF.

[4] J. Pollicino, "Bluetooth SIG unveils Smart Marks, explains v4.0 compatibility with unnecessary complexity," Engadget, 25 October 2011. [Online]. Available: https://www.engadget.com/2011-10-25-bluetooth-sig-unveils-smart-marks-explains-v4-0-compatibility-w.html.

[5] "Ember Travel Mug²," [Online]. Available: https://ember.com/products/ember-travel-mug-2.

[6] F. Islami, "A prospective study of tea drinking temperature and risk of esophageal squamous cell carcinoma," *Wiley Online Library,* vol. 146, no. 1, 2019.

[7] "Heat, Work and Energy," Engineering ToolBox, 2003. [Online]. Available: https://www.engineeringtoolbox.com/heat-work-energy-d_292.html. [Accessed 23 Mar 2020].

[8] st, "USB Power Delivery and Type-C," 2016. [Online]. Available: https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/group0/5a/b1/8e/6c/2b/0d/46/3c/Apec/files/APEC_2016_USB_Power.pdf/_jcr_content/translations/en.APEC_2016_USB_Power.pdf.

[9] Arduino, "Arduino Mega R3," Arduino, [Online]. Available: https://store.arduino.cc/usa/mega-2560-r3.

[10] QKits, "WEMOS SIMPLE WIFI BOARD WITH ESP32 MODULE!," QKits, [Online]. Available: https://store.qkits.com/wemos-simple-wifi-board-with-4mb-flash-based-on-esp-8266ex.html.

[11] Arduino, "ARDUINO ZERO," Arduino, [Online]. Available: https://store.arduino.cc/usa/arduino-zero.

[12] "ARDUINO NANO 33 BLE," Arduino, [Online]. Available: https://store.arduino.cc/usa/nano-33-ble.

[13] AliExpress, "0.91 inch 128x32 I2C IIC Serial Blue OLED LCD Display Module 0.91" 12832 SSD1306 LCD Screen for Arduino Backlight," [Online]. Available: https://www.aliexpress.com/item/32788923016.html?spm=a2g0s.9042311.0.0.27424c4dORCYlM.

[14] "Waveshare 212x104,2.13''e paper/E-Ink display HAT for Raspberry Pi 2B/3B/Zero/Zero W,Three-color: red,black,white.SPI interface," AliExpress, [Online]. Available: https://www.aliexpress.com/item/32829497666.html?spm=a2g0s.9042311.0.0.27424c4dwoQFfJ.

[15] "Dallas Semiconductor," [Online]. Available: https://store.qkits.com/assets/pdf/DS18S20.pdf. [Accessed 25 Mar 2020].

[16] "Android 4.3 Bluetooth Low Energy unstable," Stackoverflow, 25 Jul 2013. [Online]. Available: https://stackoverflow.com/questions/17870189/android-4-3-bluetooth-low-energy-unstable.

[17] "045: Bluetooth (LE) with Dave (devunwired) Smith," fragmentedpodcast, [Online]. Available: https://fragmentedpodcast.com/episodes/45/.

[18] Paweł, "RxAndroidBLE - your most powerful tool for Bluetooth Low Energy coding!," [Online]. Available: https://www.polidea.com/blog/RxAndroidBLE_the_most_Simple_way_to_code_Bluetooth_Low_Energy_devices/.

[19] G. Schorcht, "MH-ET LIVE MiniKit," riot-os, [Online]. Available: https://doc.riot-os.org/group__boards__esp32__mh-et-live-minikit.html#pinout.

[20] C. C. A. R. D. Kevin Townsend, "Getting Started with Bluetooth Low Energy," oreilly, [Online]. Available: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch01.html.

[21] Google, "Bluetooth low energy overview," Android Developers - Google, [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le.

[22] "GATT Services," Bluetooth SIG, [Online]. Available: https://www.bluetooth.com/specifications/gatt/services/.

[23] "1-WIRE PROTOCOL PDF OF DS18S20 VS DS18B20 DIGITAL THERMOMETERS," maximintegrated, [Online]. Available: https://www.maximintegrated.com/en/design/technical-documents/app-notes/4/4377.html.

[24] M. v. Welie, "Making Android BLE work — part 1," Medium, 23 Mar 2019. [Online]. Available: https://medium.com/@martijn.van.welie/making-android-ble-work-part-1-a736dcd53b02.

[25] J. Goodwin, "Charging my ThinkPad X1 Gen4 using USB-C," livejournal, 28 Jan 2017. [Online]. Available: https://laptop006.livejournal.com/59323.html.