EDWARD F. CROSS SCHOOL OF
ENGINEERING
WALLA WALLA UNIVERSITY

Thermoelectric Wrist Cuff

Eryn Hopps, Ito Perez, Natalie Smith

6/15/18

Advisors:

Dr. Larry Aamodt

Dr. Rob Frohne

Abstract

Human body temperature can vary widely, leading to potential discomfort in rooms that are too hot or too cold. Furthermore, a sickness known as Raynaud's disease -- cold extremities -- can cause actual pain if body temperature if not regulated closely enough. To combat this issue, our team is developing a thermoelectric temperature regulation device that a user can wear around his or her wrist. Our device takes advantage of the localized temperature-sensitive area on the inner wrist to control the user's perceived body temperature, making users feel more comfortable no matter the temperature of the room around them.

There are three main goals for our project. The first goal is to create a device that can be used as therapy to those who suffer from cold hands, inflammation in joints, pain in the muscles, etc. The second goal was to create a personal temperature device that would change the perceived temperature of the user. For example, make the user feel colder on a hot day or vis-versa. The third goal was to use the opportunity to save energy by reducing the use of heating and air conditioning used in personal homes and in business buildings.

Our device was designed in four different main parts, hardware, power, software to control the microcontroller, and the Android app. Natalie designed the hardware and power parts of the device. Eryn implemented the software to control the device using a microcontroller. Ito created an Android app that will allow users to control the thermoelectric wrist cuff. Together we created a working prototype that is both user friendly, effective, and safe.

All three of our goals were obtained when the project was complete. Allowing a device to warm the blood entering the hands resolved the issue of cold holds and achieved the first goal. The second and third goal was obtained through volunteers' testing the device. People reported that it was both effective in changing their perceived temperature and efficient in saving energy by not using heating or air conditioning when they were only at home for a short amount of time. The thermoelectric wrist cuff is both a successful and useful device.

Table of Contents

Table of Figures

List of Tables

Table of Equations

Acknowledgements

Thank you Dr. Rob Frohne and Dr. Larry Aamodt for advising our team during the progression of this project. Dr. Aamodt played a crucial role in shaping our project vision by motivating us to create specifications and goals before starting the project. Dr. Frohne was an excellent source of technical knowledge involving both the hardware and software sections of the project.

Also, a special thank you to Professor Kari Firestone and Professor Lucille Krull for dedicating time from busy schedules to meet with our team to give us insight on the possible effects of our device. Their contribution encouraged us to implement a variety of safety factors into our design.

1. Background/Reasons why we choose this project

The idea of creating a thermoelectric wrist cuff arose, because one of the team members, Natalie Smith suffers from Raynaud's syndrome. This syndrome causes "smaller arteries that supply blood to your skin to narrow, limiting blood circulation to affected areas" such as fingers and toes, also known as vasospasm (Mayoclinic). Randomly throughout the day no matter what the surrounding temperature is, Natalie possibly will have the effects of this syndrome occur in mainly her hands. There are several methods to encourage circulation back into her hands. One, she could massage her hands until blood begins to flow which is typically a very painful process. Another way is to place her hands under warm water for several minutes, which is inconvenient. Therefore, one of her main goals was to create a device that could warm the blood entering her hands. Specifically, creating a device that could be worn during the day at any time of the year creating a convenient and painless solution. This is when she, Eryn Hopps, and Ito Perez came up with the design to create a thermoelectric wrist cuff.

The thermoelectric wrist cuff consists of several different main parts. Natalie was given the tasks of creating the hardware of the device, Eryn's specialty is working with the "brains" or the software to control the device, and Ito was the expert at creating the app that would allow the user to communicate with the device and control the temperature pulses as the user so desired. Figure 1 shows a block diagram of the thermoelectric wrist cuff.

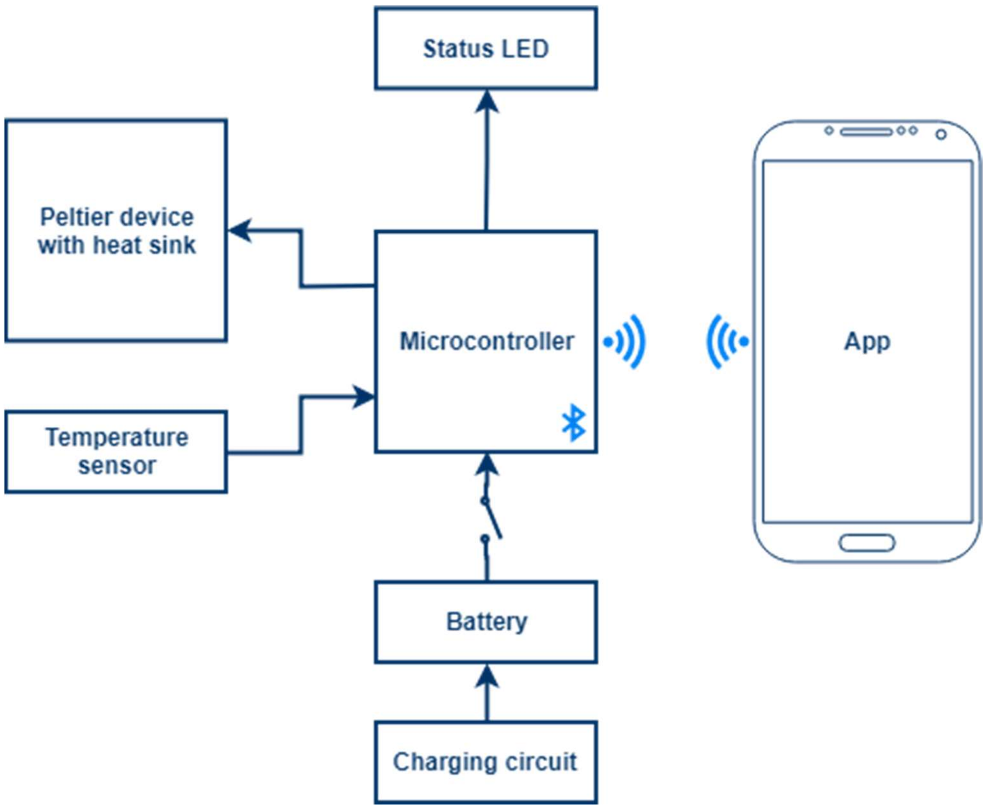


Figure 1. Block diagram of the thermoelectric wrist cuff

There are several locations on the body that are known to be used as localized sensations (Barry Green 1997, pg 331). These are locations around the body that are highly sensitive to temperatures and may affect the body. Some of these localized sensation locations include the ankles, back of the neck, wrists, and several more (Barry Green 1997, pg 337). The most convenient localized temperature location that would most greatly benefit those with Raynaud's syndrome is the underside of the wrist where all the blood enters and leaves the hand. Warming the blood as it enters the wrist will open the blood vessels in the fingers resolving the issue for Raynaud's syndrome. This was tested to be true as discussed further in the *Results* section.

In addition to wanting a device that can assist those who suffer from Raynaud's disease, the team believed that this device could be used for other therapy. If there is pain in the wrists, switching between applying hot and cold temperatures helps relieve the pain and reduce swelling. This is an excellent health device that may assist those requiring such therapy.

A second goal was to create a device that could be used as a personal temperature device. The idea is analogous to placing a cold towel on your body to cool down when it is hot outside or holding a hand warmer to warm up when it is cold outside. The same idea can be applied to the use of this device. It will warm or cool the blood as it leaves the wrist and flows to the rest of the body making the person feel warmer or colder.

Finally, our third goal is to reduce the use of air conditioning and heating used in buildings and homes. As this will assist those to regulate their temperature, it will allow people to not constantly adjust their house temperature to feel more comfortable. In addition, companies can have happier employees that feel comfortable at work rather than constantly feeling too cold or warm. Reducing the use of air conditioning and heating may overall save energy and money for the user and companies.

2. Research on localization/interview quotes

Several interviews were conducted to determine the feasibility and affects this project would have on a user. To begin, we needed to determine the effects of localizing both hot and cold temperatures in one location. We interviewed several medical professionals.

One of our main concerns was to understand how much heat or cold could be applied in one location before harming the user. During an interview with Professor Kari Firestone Ph.D. (Registered Nurse and Certified Nurse Specialist), she informed us that having too much heat or cold in one location produces similar results. The skin will begin to get red, eventually turning into first degree burn, then second degree and so on. Another interview conducted with Professor Lucille Krull Ph.D. (Registered Nurse) informed us that it is difficult to specifically identify a cold temperature where humans begin to show symptoms of harm being done to the skin. This is because everyone is so different one example being Natalie's Raynaud's syndrome.

We asked Dr. Firestone if she believed it would be better to cycle the heat and cold temperatures versus having the temperatures constantly on the persons' skin. She provided insight to this situation by informing us that constantly having heat or cold on the skin may eventually cause the skin to redden which is the first sign of burns but not necessarily burns. Another factor against constant temperature on the skin is that the person may get used to feeling the temperature and the use of wrist cuff will then be defeating its purpose. Therefore, from her advice we decided to pulse the hot or cold temperature to create the effect of feeling hotter or colder. The amount of heat that we decided to pulse is discussed later.

To end the interview Dr. Firestone discussed with us the advantages and disadvantages of a device like this. The only disadvantage of the device that she told us is that it could possibly injure the user if they were not careful how much heat or cold their bodies could handle. An example of this was shown during the testing of our device. Our professor suffered from a stroke a few years ago leaving him to lack feeling in his left arm. We placed our thermoelectric wrist cuff on his left wrist and turned it on to as cold as we were planning to limit the device. He was unable to tell if it felt cold or warm. So, we made the device put out as much heat as we were planning to allow it to output. Again, he was unable to determine whether it was hot or cold. We then placed our device on his other wrist and he was able to feel it hot or cold as we expected him to. This is a great example of how the device could potentially be dangerous for those who are unable to feel heat or cold. This disadvantage was also a concern of Dr. Krull. With this information conveyed to us, we then implemented several safety features which will be discussed later throughout the paper.

While Dr. Firestone pointed out this disadvantage, she did identify several advantages. She did her PHD research in cold therapy techniques. She believed that it would be possible to use our device as therapy. Pulsing both hot and cold temperatures can reduce the swelling of an injury. Besides therapy of swelling, she believed that it would assist those who suffer from Raynaud's syndrome. The location of the Peltier is in the optimal location on the wrist to warm the blood that enters the hand. Therefore, warmer blood will cause the blood vessels in the hand to open creating circulation and therapy for those who suffer from Raynaud's syndrome. Overall, the advantages of our device outweighed the disadvantages according to the professors that we interviewed.

3. Hardware

There are several key important parts involved in the thermoelectric wrist cuff. This includes a safe way to transmit hot or cold temperature to the user, a method to power the device including safety features to protect the user, and a way to make the device user friendly. The following sections will discuss the methods used to decide why we chose to use each of the components.

3.1. Thermal generation

Two options for thermal generation were considered, one of which was zeolite. Zeolite is a mineral used as an absorbent and catalysts (Wikipedia). One characteristic it has is that when water is added to it, a chemical reaction occurs causing it to warm up quickly. The zeolite will stay hot until all the water has evaporated from the beads of zeolite. It can be reused simply by adding more water. The advantage of using zeolite, is that it is efficient requiring no electricity or manmade energy to run it. It only requires water which is typically an accessible resource. Another advantage of zeolite is that it gets hot fast and stays hot for a long time. While there are many advantages, there are several disadvantages. One of the biggest disadvantages is that it cannot be cold, only hot. Also, it is not possible for the user to control how hot the zeolite will get. Both disadvantages defeat several main objectives of this project.

The other thermal generation that we considered was a thermoelectric Peltier device. The Peltier device is made of thermal couples connected in series. In 1821, a man named Thomas Seebeck discovered that "a circuit made from two dissimilar metals, with junctions at different temperatures would deflect a compass magnet" (Caltech). Magnetization arises from electrical current as is proven in Ampere's law. Jean Peltier was a French watchmaker and physicist who discovered that depending on the direction flow running through two dissimilar metals, heat could be removed or added to the junction of the metals (Caltech). Based on this discovery, a device called a Peltier device was created, Figure 2.

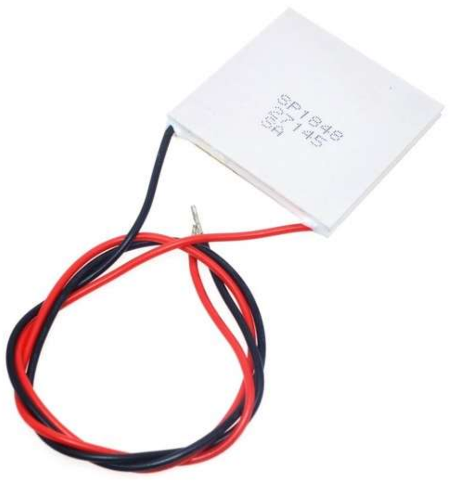


Figure 2. Thermoelectric Peltier

We were then able to apply the knowledge from Jean Peltier to manipulate a Peltier device to work as we desired. As shown in Figure 3 (A), as the current flows through the red wire, into the Peltier device, and out the black wire, one side of the Peltier device will get hot while the other side will get cold. Likewise, in Figure 3 (B), running the current in through the black wire and out the red wire will cause the hot and cold sides to switch.

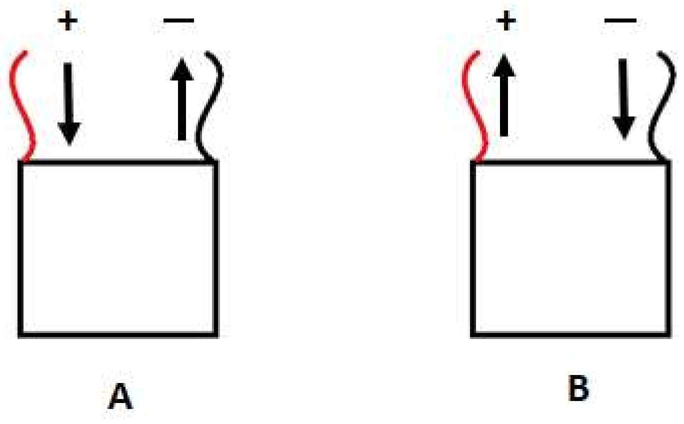


Figure 3. How a Peltier Works

The Peltier device works well as the thermal generation for this project for many reasons. One of the greatest reasons, is that the thermal generation is both reversible (can be hot or cold) and controllable with a microcontroller. As described later, a microcontroller can be used to regulate the amount of current flowing into the Peltier device. The more current the hotter or colder the Peltier device will get. Therefore, two main objectives of this project, to make the thermal generation reversible and user friendly, is possible with the Peltier device. The main disadvantage of the Peltier device is that it is not efficient. With the options available, this disadvantage did not detour us from using it for our thermoelectric wrist cuff.

3.2. Heat sink

To increase the efficiency of the Peltier device, a heat sink is used to dissipate heat when the user wants to be cold. This is an example of the first law of thermodynamics, if we did not have a way to dissipate heat, most of it would travel into the colder side of the Peltier device if not distributed out into the surrounding air (NASA). This would make it harder for the user to feel colder if they so desired.

There were several criteria when choosing a heat sink for our project. We needed a heat sink that was economical, light weight, and small. Typically heat sinks are made from either aluminum, brass, copper, or steel (Design World). Aluminum is more economical, therefore we decided to have an aluminum heat sink only costing $3.34. Since it is made from aluminum, it is light weight. The size of our heat sink is the same size as the Peltier device (30 x 30 x 10 mm). Having a heat sink larger than the Peltier device would use unnecessary space and a smaller heat sink would be less efficient than one that covered the entire Peltier device.

3.3. Temperature sensor

Sensing the temperature will be used as a safety feature to restrict the thermoelectric wrist cuff from getting too hot or too cold, possibly causing harm to the user. If the microcontroller detects that the temperature is surpassing the specified limitations, it will automatically shut down the device.

An analog temperature sensor was considered because it is cheap ($0.94) and small. It has a wide range of operating temperatures from -50 to 150 degrees Celsius. A thermistor was also considered. The thermistor that we considered is smaller than the analog temperature sensor, and easy to measure temperature from. It was also cheaper at $0.75 and still small enough that it was unnoticeable when attached to the Peltier (see *Testing* section for more details about placement of the thermistor).

To sense the temperature, we placed the thermistor in a voltage divider configuration (see Figure 4). We then would measure the voltage current through one of the pins on the Nordic microcontroller (Vo in Figure 4).

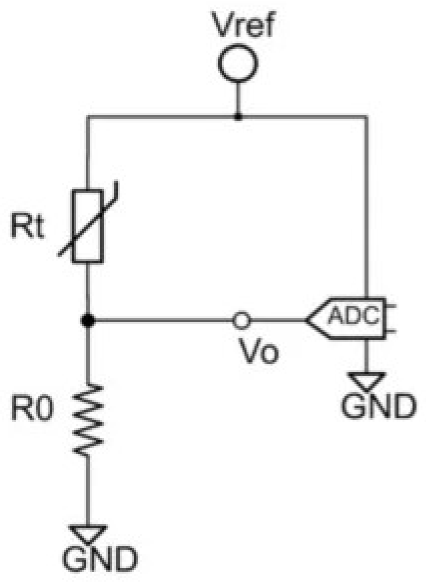


Figure 4. Thermistor voltage divider

The thermistor can handle a resistance from 3.3 to 470 kΩ. Using voltage division (Equation 1), calculations as shown below were completed to determine that the range of voltage for the thermistor are 3.6229 to 0.9181 volts.

$$V_o = V_{ref} * \frac{R_t}{R_t + R_0}$$

Equation 1. Voltage Division

Where Rt = thermistor measurement, R0 = 10,000 ohms, Vref = 3.7 volts, and Vo = measured voltage across the thermistor. For calculating the highest voltage that can be across the thermistor, we will use 470k ohms as R1.

$$V_o = 3.7\ V * \frac{470k\Omega}{470k\Omega + 10k\Omega} = 3.6229\ V$$

For calculating the lowest voltage that can be across the thermistor, we will use 3.3k ohms as R1.

$$V_o = 3.7\,V * \frac{3.3k\Omega}{3.3k\Omega + 10k\Omega} = 0.9181\,V$$

The temperature that the thermistor measures can be calculated with Equation 2. Given by the data sheet of the thermistor selected for this project, A_1 = 3.354E-03, A_1 = 3.354E-03, A_1 = 3.354E-03, and A_1 = 3.354E-03 are the constants used in Equation 2.

$$T_R = [A_1 + B_1 \ln\left(\frac{R}{R_{ref}}\right) + C_1 \ln^2\left(\frac{R}{R_{ref}}\right) + D_1 \ln^3\left(\frac{R}{R_{ref}}\right)]^{-1}$$

Equation 2. Calculated thermistor temperature

3.4. Power source

The criteria to choosing a power source was something small, inexpensive, light weight, and able to provide the necessary current to power the Peltier device. We created a table to compare the specifications of all the power sources that we considered, see Table 1.

Type	**Model Number**	**Current**	**Voltage**	**Size**	**Cost Each**	**Rechargeable**	**Quantity Needed**	**Lasting Time**	**Total Cost**
-	-	mA	V	mm	$	Y/N	-	min	$
Coin Cell	CR2032	250	3	20 x 3.2	1.95	N	3	25	5.85
Coin Cell	CR2450	110	3.6	24.5 x 5.2	2.95	Y	6	11	17.7
Lithium Ion	DTP603443	850	3.7	44.45 x 34.79	9.95	Y	1	85	9.95
Lithium Ion	DTP605068	2000	3.7	49.2 x 68.8 x 5.6	13	Y	1	200	13
Lithium Ion	DTP502535	400	3.7	20 x 11 x 3	4.5	Y	2	40	9

Table 1. Battery power comparison

Type	**Model Number**	**Current**	**Voltage**	**Size**	**Cost Each**	**Rechargeable**	**Quantity Needed**	**Lasting Time**	**Total Cost**
-	-	mA	V	mm	$	Y/N	-	min	$
Coin Cell	CR2032	250	3	20 x 3.2	1.95	N	3	25	5.85
Coin Cell	CR2450	110	3.6	24.5 x 5.2	2.95	Y	6	11	17.7
Lithium Ion	DTP603443	850	3.7	44.45 x 34.79	9.95	Y	1	85	9.95
Lithium Ion	DTP605068	2000	3.7	49.2 x 68.8 x 5.6	13	Y	1	200	13

Lithium Ion	DTP502535	400	3.7	20 x 11 x 3	4.5	Y	2	40	9

Table 1. Battery power comparison

The lightest power source that we considered was a coin cell battery. While coin cells are light and small, it would take several to provide enough amps to power our device longer than an hour. Therefore, the next battery that was considered was a small lithium ion battery. The lithium ion battery can come in a variety of sizes. We chose to use one that was 0.850 Ah and 3.7V. This is because it is small and thin at a size of 1.69 x 1.34 x 0.35 inches. It also was light weight only being 20 g. And finally, it provided enough amperes to power our device at its full capacity for more than four hours.

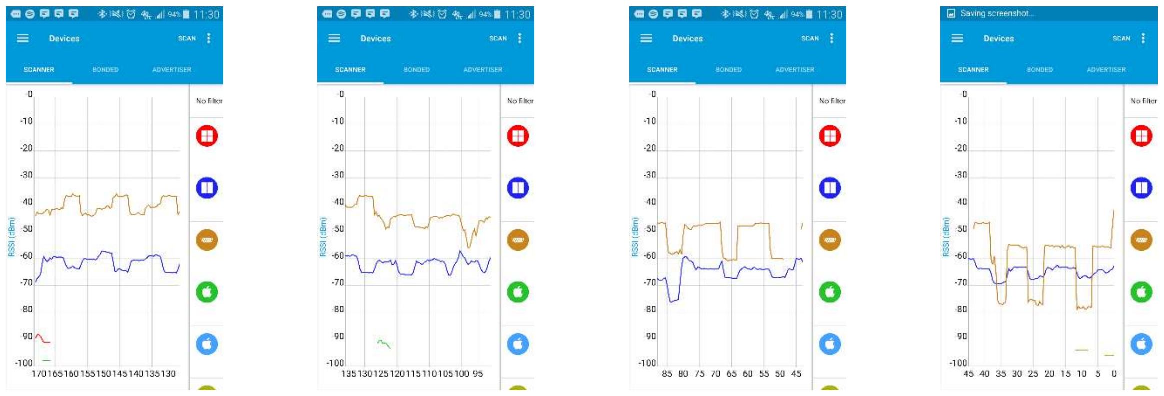
As a safety feature, we incorporated a manual switch that will allow the user to turn on or off the device. If the device disconnects to the app or if it begins to get too hot or cold for the user, they have an easy way of cutting off power to the device.

Another feature that we wanted to incorporate into our board was a way to determine whether the device was on and sending heat to the user or sending cold to the user. Therefore, a Red Green Blue (RGB) LED was used. When the device is on and not pulsing any temperature to the user, the LED will pulse a green light. When heat is transferred to the user, the LED will pulse red. When the device sends cold pulses to the user, the LED will pulse blue.

3.5. Printed circuit board

To test our device, we created a preliminary program using an embedded board. The results are discussed in the *Testing* section. Once we tested our device successfully, we decided to create a compact version of it which required designing a printed circuit board (PCB). One of the struggles that we came across was deciding which way the microcontroller was to be placed as it had through-hole connectors and could be placed upside down or right side up.

To determine whether or not we could successfully orient the microcontroller upside down, we tested the RSSI level received from the microcontroller by a phone running Nordic Semiconductor's nRF Connect app. We placed another PCB over the module at varying heights, covering the antenna. Figure 5 shows the results of this test for four different levels of antenna blockage.



No blockage

Small blockage (~1cm)

Medium blockage (~0.75cm)

Full blockage (touching)

Figure 5. RSSI testing with RedBearLabs BLE Nano v2 module

The brown line represents the RSSI level of the RedBearLabs module, and the blue line represents the RSSI level of another nearby device. As shown in the figure, the RSSI level of the module oscillates regularly regardless of the amount of blockage in front of the antenna. As the blockage increases, the RSSI level gets significantly lower. After performing this test, we decided the difference in signal strength was too great and chose to orient the module right side up.

Once we chose all the parts that were needed for the device and how we wanted to connect all of them together, we created a schematic (see Appendix 1). The next step was to create a PCB layout, for which the program KiCad was used to place the components in a desired layout and connecting them together. The final PCB layout is shown in

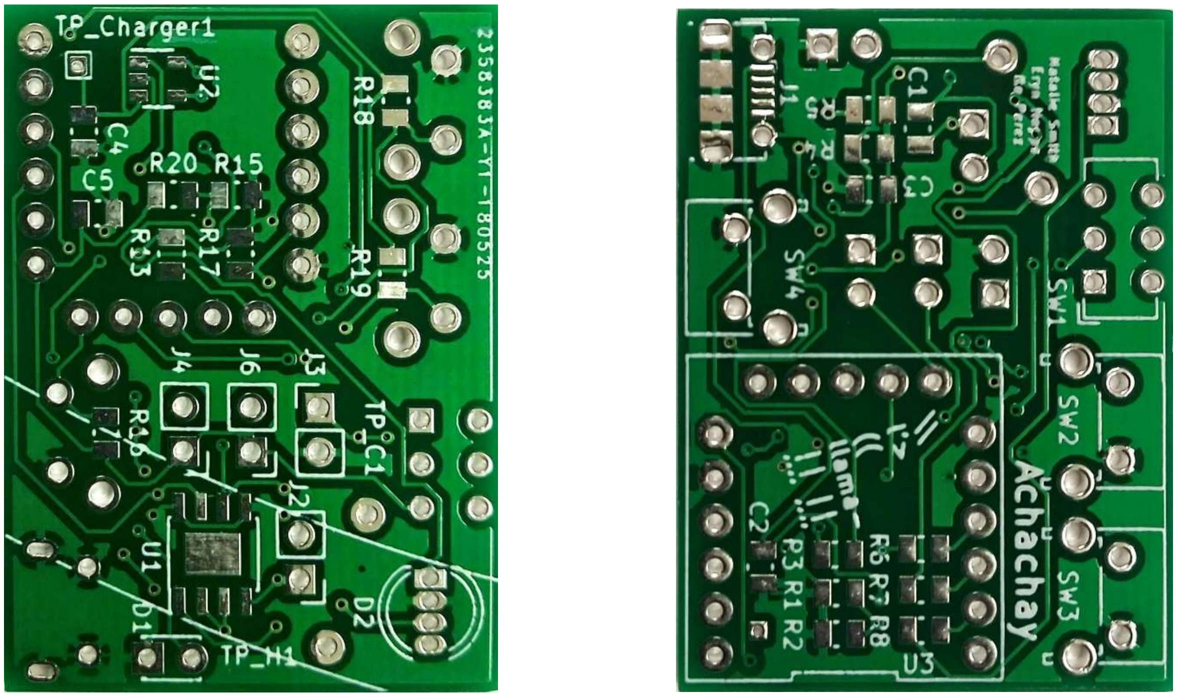


Figure 66.

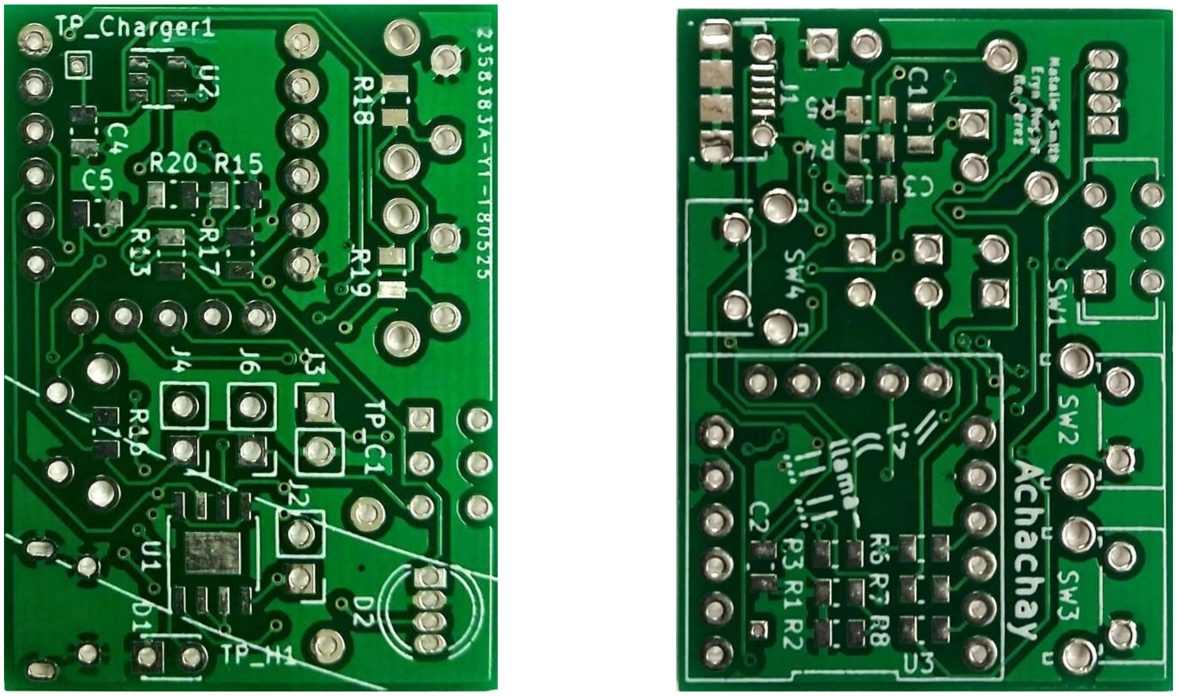


Figure 6. PCB layout (top view right and bottom view left)

This PCB was used as our final design as it worked as desired without any flaws or without needing further changes,

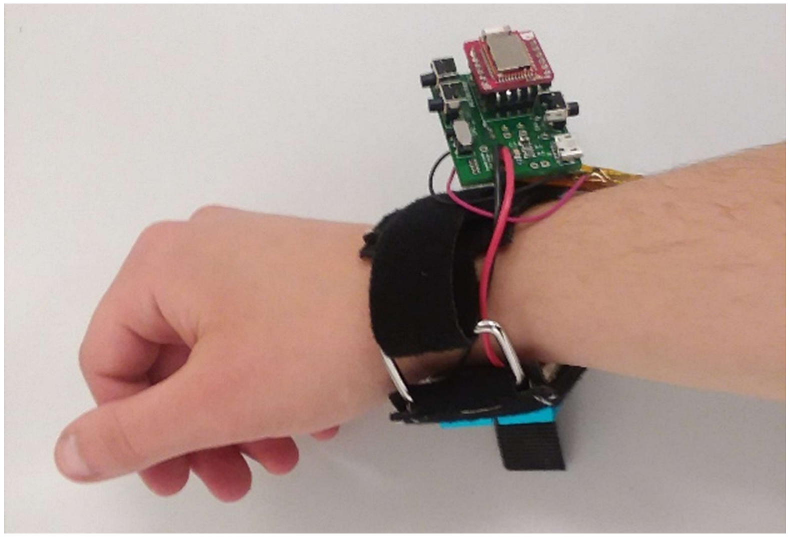


Figure 7. Buttons were also added to the original design to use the device if the phone is inaccessible. The PCB is smaller than the battery allowing it to fit on top of the wrist without further restricting movement.

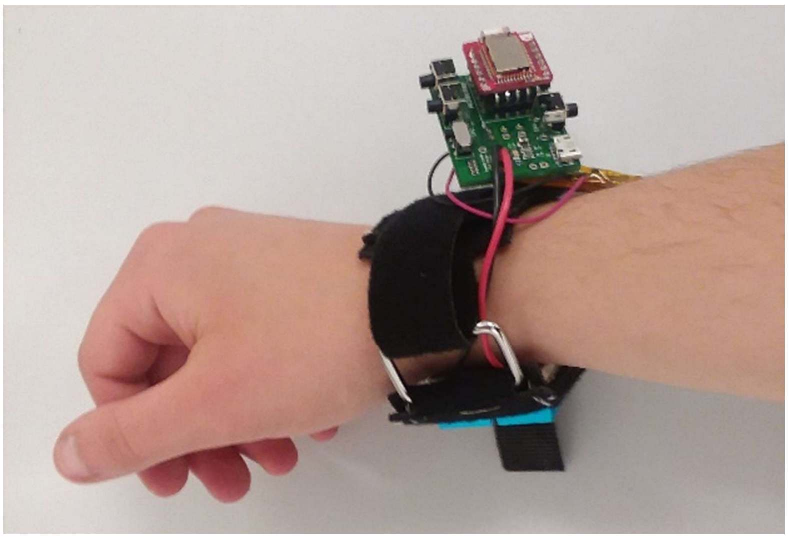


Figure 7. Final design

4. Firmware

To operate our device, we must equip it with a microcontroller. This microcontroller receives commands from the user's app, controls current flow to and from the Peltier device, takes input from the onboard temperature sensors, sends their readings to the app, and controls the status LED.

4.1. Choosing a communication protocol

To establish a link between the device and the app, our device uses a Bluetooth connection. Because of Bluetooth's ubiquity in today's market—100% of mobile devices shipped in 2018 will include Bluetooth functionality ("Build Your Product")—we choose it for our device over other communication protocols like ANT and Zigbee.

Bluetooth comes in two varieties: Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR) and Bluetooth Low Energy (BLE). BR/EDR, also known as Bluetooth Classic, is the main variety, usable for everything from small information packets to high-quality audio and video streaming. Bluetooth Low Energy, also known as Bluetooth Smart, is an offshoot of BR/EDR used only for devices that do not need to send large quantities of data. A device using BLE consumes only a fraction the power of one using BR/EDR. The Bluetooth Special Interest Group (Bluetooth SIG), Bluetooth's governing body, gives a comparison of BR/EDR and BLE radios, which is reproduced in part in the table below.

	Bluetooth Low Energy	**Bluetooth Basic Rate/Enhanced Data Rate**
Optimized for	Short burst data transmission	Continuous data streaming
Frequency band	2.4GHz ISM band (2.402-2.480 GHz utilized)	2.4GHz ISM band (2.402-2.480 GHz utilized)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel usage	Frequency-hopping spread spectrum (FHSS)	Frequency-hopping spread spectrum (FHSS)
Modulation	GFSK	GFSK, $\pi/4$ DQPSK, 8DPSK
Power consumption	~0.01x to 0.5x of reference (depending on use case)	1 (reference value)

Table 2. Comparison of Bluetooth Low Energy and Bluetooth BR/EDR ("Radio Versions")

From this table, we see that BLE provides all the functionality we need and consumes the least power. We will cover more specifics of designing using BLE in upcoming sections.

4.2. Choosing a microcontroller

Knowing that our desired communication protocol is BLE, we select a BLE-capable microcontroller chipset. Our device uses a Nordic Semiconductor nRF52832 microcontroller, which provides all the functionality it needs (as covered in a later section). Our design process for choosing the microcontroller is outlined in the following sections.

Separate or integrated BLE?

We know our device's chipset must have two main capabilities: general processing and BLE communication. To achieve this, we can select any microcontroller and any dedicated Bluetooth chip and configure them to communicate with each other. Alternatively, we can select a microcontroller with integrated Bluetooth capability. Because of size and power constraints, we choose a combined BLE-capable microcontroller.

Chip or module?

To install a Bluetooth-enabled microcontroller in a device, the designer has two options. One option is to purchase the microcontroller as a standalone chip and install it directly in the device. This requires the designer to create an antenna over which the device can communicate, ensure that the antenna is situated and wired correctly with respect to the microcontroller, and for commercially available devices, obtain FCC and Bluetooth SIG certification. Certifying a custom-designed device with the Bluetooth SIG can cost $8000 (Rossi 2017). Although we are no longer choosing to commercialize our device, we initially wanted to have the option to do so; thus, certification would be a large obstacle.

The other option is to purchase a premade module containing both the microcontroller and an antenna. These modules do not allow the designer control over the antenna design and layout. Choosing to package the chip inside a premade module can also restrict the functionality of the device, as the module may or may not break out all of the microcontroller's pins. However, modules do include pre-made antennas, and they come pre-certified by the FCC and the Bluetooth SIG.

Because of the time and money required to design an antenna and receive certification, we choose to implement our microcontroller using a pre-designed module. Since our device requires relatively few peripherals, we accept the possibility that the module will not provide access to all available pins, and we stipulate that the module we choose must provide all the necessary functionality for our design.

Bluetooth version?

Bluetooth Low Energy was first released as part of Bluetooth 4.0; all following versions include the capability as well. As of version 5.0, however, Bluetooth has also included LE Secure Connections, a more secure communication protocol. Since our device will be used in personal health applications, we aim for the highest security possible. Furthermore, to postpone obsolescence, it is preferable to include compatibility for the most recent technology available. Thus, we prefer a chip that is capable of supporting not only current versions of Bluetooth but also Bluetooth 5, if possible.

Table 3 shows our final requirements for the chipset.

Option	**Decision**
BR/EDR or BLE?	BLE
Separate or integrated BLE?	Integrated
Chip or module?	Module
Bluetooth version?	At least 4.0; 5.0 capability preferred

Table 3. Requirements for microcontroller chipset

Microcontroller options

Our most major concerns when choosing a microcontroller are Bluetooth version, amount and quality of available online support, development environments supported, and power consumption. As previously stated, we prefer chips that are Bluetooth 5 capable. We would also like to develop using Keil µVision 5 (covered in an upcoming section) as well as Bluetooth Developer Studio, a development tool created by the Bluetooth Special Interest Group to determine appropriate profiles, services, and characteristics for Bluetooth devices. To decrease power consumption, we prefer chips that have the lowest transmit, receive, and sleep currents possible. Finally, we prefer chips that have a wide user base and plenty of documentation. Our rankings of the viable solutions we found are given in the table below.

Chip	**Bluetooth**	**Keil?**	**BDS?**	**Current (TX, mA)**	**Current (RX, mA)**	**Current (sleep, uA)**	**Support? (1-5)**
CC2540	4.0	no	yes	24	19.6	0.9	5
BlueNRG-1	4.0	yes	no	8.3	7.7	0.9	5
nRF52832	4.2/5.0	yes	yes	5.3	5.4	0.3/ 0.7/ 1.9	5
DA14680	4.2	no	no (own equivalent)	4.9	4.9	0.6	3
BLE113	4.0	no	yes	18.2	14.3	0.4	4
RL78/G1D	4.1? 4.2?	no	yes	4.3	3.5	0.3	3

Table 4. Comparison of microcontroller options

Based on these rankings, the Nordic Semiconductor nRF52832 seems like an optimal choice. Unlike any other chip on our list, the nRF52832 operates with Bluetooth 4.2 and is hardware capable of operating with Bluetooth 5 for further upgrades, and it allows development in Keil µVision and Bluetooth Developer Studio. Its operating currents are among the lowest in the group, it is widely used by developers, and it has thorough, detailed documentation. For these reasons, we choose the nRF52832 as our microcontroller.

However, to avoid FCC and Bluetooth SIG certification costs, we must still choose a module in which to implement it. Our chief concerns here are the module's physical size and the number of pins it makes available—we would like the device to fit comfortably on the wrist, and we must have enough pins to complete our tasks.

To meet these criteria, we have chosen the RedBearLabs BLE Nano v2 module to contain our nRF52832 chip. In addition to coming with a built-in antenna and FCC and Bluetooth SIG certification, the BLE Nano v2 is approximately the size of a quarter and breaks out 11 usable pins. While this is a small number, it is sufficient for our needs, which we discuss in an upcoming section. Furthermore, since the BLE Nano v2 is a development board, we can develop and test with it without mounting it on a custom PCB. This way, our development board is also the board we use in our final prototype.

Figure 8. Nordic Semiconductor nRF52832 ("nRF52832")

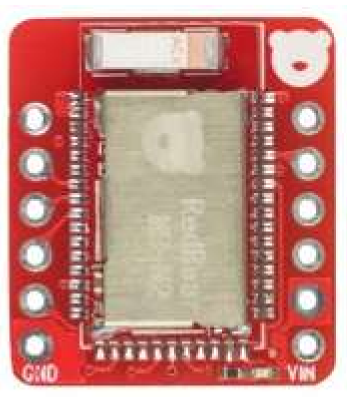


Figure 9. RedBearLabs BLE Nano v2 ("BLE Nano v2 (No Header)")

4.3. Interfacing with app and peripherals

Determining a pinout

To communicate with and control all the cuff's other components, our microcontroller exchanges data with the Android app, controls the current flowing to the Peltier, gets periodic readings from the temperature sensor, and operates the status LED. To do these things, the microcontroller uses the following modules, pins, and capabilities:

- Send data to and receive data from the Android app
    - Bluetooth Low Energy
    - Serial communication module (e.g. a UART)

- Control the current flowing to the Peltier
    - Pulse-width modulation capability
    - Two GPIO pins

- Get periodic readings from the temperature sensor
    - Analog-to-digital converter
    - Timer
    - One GPIO pin with ADC functionality

- Operate the status LED
    - Three GPIO pins

- Battery sensing
    - Analog-to-digital converter
    - One GPIO pin

- Manual temperature adjustment
    - Two GPIO pins

- Bluetooth pairing
    - One GPIO pin

The BLE Nano v2's biggest selling point—its small size—is also its biggest disadvantage: the module breaks out very few of the nRF52832's pins. For our purposes, however, it breaks out just enough. The nRF52832's UART can be moved to any set of GPIO pins. Since we will only be transmitting serial data over BLE, we have no need to access the UART through physical pins. Thus, by moving the UART to pins that are not broken out on our module, we can instead use pins 28, 29, 30, and 2 for other peripherals. We use 28, 29, and 30 for the red, green, and blue LED pins and 2 for the temperature sensor.

In addition, we have broken out the bottom five pads (pins 3, 6, 7, 8, and 21) for extra features. These pins allow us to connect another temperature sensor, monitor the battery level, control temperature using physical buttons, and initiate Bluetooth pairing using another physical button. These buttons and sensors are physically connected to the board through the five extra pads, but their functionality has not been added in firmware. This is a task for further work.

Our final pinout is shown in the figure on the following page.

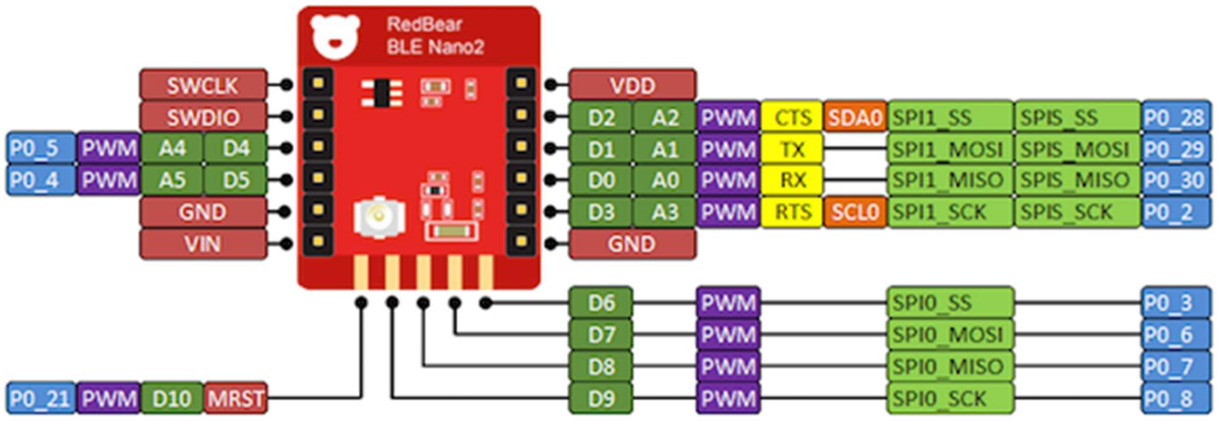


Figure 10. BLE Nano v2 pinout ("Nano2 Pinout")

Peripheral	**Function**	**Pins**
Android app	BLE UART	N/A (wireless)
Peltier	Pulse-width modulation	P0_4, P0_5
Temperature sensor(s)	Analog-to-digital converter Real-time counter	P0_2, P0_3
Status LED	General-purpose I/O	P0_28, P0_29, P0_30
Battery sensing	Analog-to-digital converter Real-time counter	P0_6
Manual temperature adjustment	General-purpose I/O	P0_7, P0_8
Bluetooth pairing	BLE General-purpose I/O	P0_21

Table 5. Functions and pins needed by the nRF52832

Setting up the development environment

Because it is a major platform, because we have prior experience with it, and because Walla Walla University has purchased full licenses to develop with it, Keil µVision 5 is the integrated development environment we use to program our device. Keil µVision comes as a part of Keil's MDK-Core development kit for Windows operating systems and cannot be run on other platforms. (A free version of the software is available at Keil's website under Product Downloads -> MDK-Arm or at this URL: http://www.keil.com/mdk5/core.) Using µVision, we can flash the Nordic SoftDevice to our chip and begin programming it.

Alternatively, the nRF52832 can also be programmed using open-source tools ("nRF52832"). Since Nordic's software development kit supports the GNU G++ compiler, any development environment powered by this compiler can be used with our device.

Using libraries, drivers, and examples

Along with the nRF52832, Nordic Semiconductor has released a software development kit (SDK). The SDK contains drivers and libraries to provide the developer with a high-level interface to the nRF52832's hardware. For this project, we are using version 14.2.0 of Nordic's SDK. This SDK provides the following relevant libraries, drivers, and examples:

Library, driver, or example	**Function**
ble_app_uart	Demonstrates use of the Nordic UART Service
low_power_pwm	Demonstrates use of Nordic's low-power PWM functionality
rtc	Demonstrates use of the real-time counter

Table 6. Libraries, drivers, and examples found in the SDK

In addition to the SDK, we have also used another example program to guide us as we operate our nRF52832. This program, released by NordicPlayground on GitHub, demonstrates a low-power use of the nRF52832's successive approximation analog-to-digital converter (SAADC).

Library, driver, or example	**Function**
saadc_low_power	Demonstrates low-power use of the SAADC

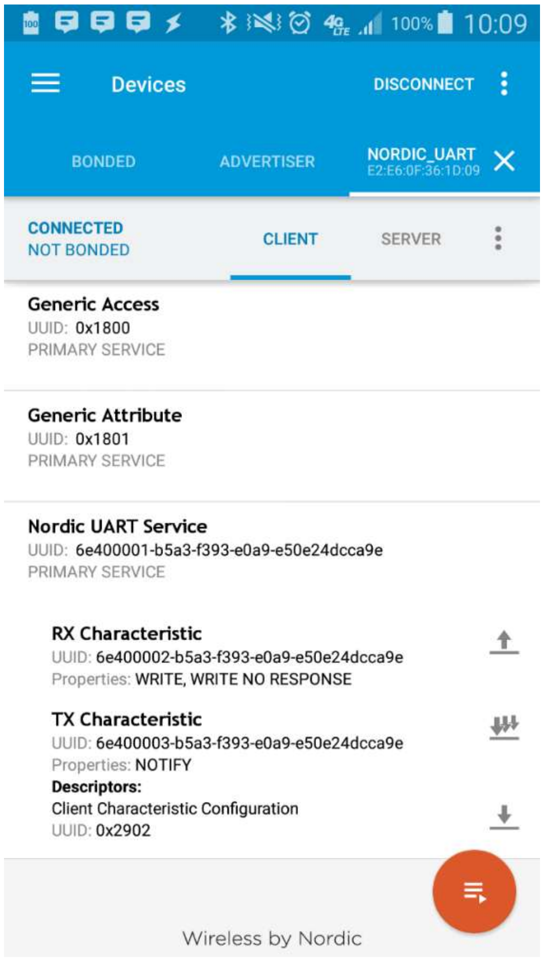
Table 7. Libraries, drivers, and examples found on GitHub

The SDK is configured to work with Nordic's default development board. However, by including a new board definition header file, we can adapt it to work with any other board we choose, so long as we know the correct pinout for the board. After creating a nanov2.h board description file, our module integrates with the SDK.

Communicating with the app

To communicate with the Android app, we use the nRF52832's built-in universal asynchronous receiver-transmitter (UART). The UART utilizes a serial communication method, meaning that we can transmit one character at a time between the cuff and the phone.

Included in Nordic's SDK is a proprietary service, called the Nordic UART Service (NUS), that enables the nRF52832 to operate its UART over BLE. This service integrates the functionality of the SDK's UART driver into a Bluetooth peripheral service. The service has two characteristics: a transmit (TX) characteristic and a receive (RX) characteristic. The TX characteristic has a NOTIFY property which allows it to notify the client when the peripheral transmits something. The RX characteristic has both WRITE and WRITE NO RESPONSE properties, which allow it to receive transmissions and write them to the peripheral's UART. Shown is a screenshot taken in Nordic's nRF Connect app that shows the Nordic UART Service and its characteristics.



Generic Access
UUID: 0x1800
PRIMARY SERVICE

Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Nordic UART Service
UUID: 6e400001-b5a3-f393-e0a9-e50e24dcca9e
PRIMARY SERVICE

RX Characteristic
UUID: 6e400002-b5a3-f393-e0a9-e50e24dcca9e
Properties: WRITE, WRITE NO RESPONSE

TX Characteristic
UUID: 6e400003-b5a3-f393-e0a9-e50e24dcca9e
Properties: NOTIFY
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

Figure 11. Nordic UART Service in the nRF Connect app

The app implements a similar UART (discussed in the App section) to send to and receive from the nRF52832. The data sent back and forth from the app to the cuff, and vice-versa, is shown below:

App to cuff	**Cuff to app**
• Temperature setting ○ Data sent at 115,200 baud ○ Data sent whenever user changes relevant setting ○ Sent as a string ○ Ranges from 0 to 20, where 0 is maximum hot, 20 is maximum cold, and 10 is off	• Temperature sensor data ○ Data sent at 115,200 baud ○ Data sent on RTC interrupt ○ Sent as a string ○ Voltages range from 0.9V to 3.6V ○ Interpreted by app

Table 8. Data exchange between app and phone

Whenever the cuff's Nordic UART Service receives a transmission, the NUS handler is triggered. This handler takes the data received over NUS and stores it in the UART's first-in-first-out (FIFO) buffer. To Nordic's UART service, we have added a function that retrieves the data from the FIFO buffer and stores it in a variable (aptly named received_data). This allows us to manipulate the data.

When the NUS handler is triggered, we typecast the received data to a signed integer then calculate the corresponding duty cycle as follows:

$$\text{dutycycle} = (\text{receiveddata} - 10) \times -10$$

We can then use this duty cycle variable to control the Peltier module.

Currently, any Bluetooth-capable master device, such as a phone or tablet, is able to connect to our device. Our device is hardware capable, using a built-in button connected to a GPIO pin, of implementing a more secure Bluetooth pairing mechanism, and this is an item for future work.

Controlling the Peltier

As discussed previously, the Peltier module is operated by flowing current into one lead and out of the other. Its temperature is dependent on the amount of power it receives, which according to the power law $(P = VI)$ can be controlled in two ways: manipulating voltage and manipulating current. We choose the latter option: we operate the Peltier by sending varying amounts of current through it. By reversing this current, we can reverse the Peltier—the hot side becomes cold, and vice-versa.

One of the simplest ways to control current flow using a digital system is pulse-width modulation (PWM). Pulse-width modulation is a method of controlling analog devices using digital signals. To use pulse-width modulation, we send a square wave—a periodic signal that alternates high and low—to our device. The amount of time this square wave is high is called its pulse width. Dividing this pulse width by the wave's total period gives us its duty cycle—the proportion of time the square wave is high. Figure 12 demonstrates the relationship between pulse width, period, and duty cycle.

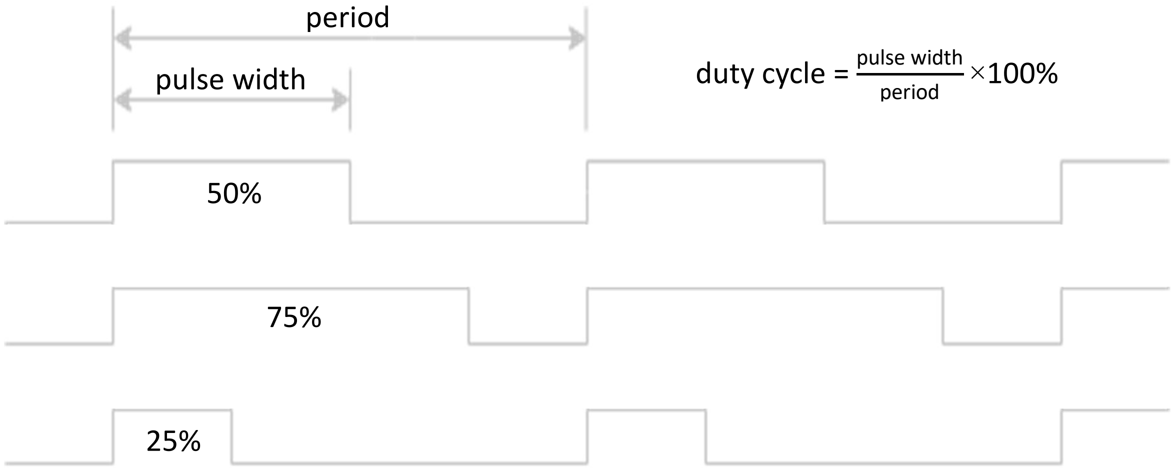


Figure 12. Pulse-width modulation

From the figure, we see that increasing the duty cycle of a square wave means increasing the amount of time it is high, and vice versa. By doing this, we are able to control the amount of average current flowing to a device: if the duty cycle is 50%, we are sending 50% of our maximum average current to the device. In this way, pulse-width modulation lets us control how hot or cold our Peltier device gets: sending more or less average current means the temperature gets more or less intense.

Since the Peltier operates by flowing current in two different directions, we must be able to source current from either side. Using the Nordic drivers, we generate two separate PWM channels, one for each pin. Ordinarily, using PWM requires a high-frequency clock for accuracy. However, to conserve power, we choose Nordic's low_power_pwm library, which uses the low-frequency clock (covered in more detail in the section entitled "Reading the temperature"). We configure each channel with an initial duty cycle of 0.

```
static void pwm_init(void)
{
    uint32_t err_code;
    low_power_pwm_config_t low_power_pwm_config;

    APP_TIMER_DEF(lpp_timer_0);
    low_power_pwm_config.active_high    = true;
    low_power_pwm_config.period         = 100;
    low_power_pwm_config.bit_mask       = HOT_PIN;
    low_power_pwm_config.p_timer_id     = &lpp_timer_0;
    low_power_pwm_config.p_port         = NRF_GPIO;

    err_code = low_power_pwm_init((&low_power_pwm_0), &low_power_pwm_config, pwm_handler);
    APP_ERROR_CHECK(err_code);
    err_code = low_power_pwm_duty_set(&low_power_pwm_0, 0);
    APP_ERROR_CHECK(err_code);

    APP_TIMER_DEF(lpp_timer_1);
    low_power_pwm_config.active_high    = true;
    low_power_pwm_config.period         = 100;
    low_power_pwm_config.bit_mask       = COLD_PIN;
    low_power_pwm_config.p_timer_id     = &lpp_timer_1;
    low_power_pwm_config.p_port         = NRF_GPIO;

    err_code = low_power_pwm_init((&low_power_pwm_1), &low_power_pwm_config, pwm_handler);
    APP_ERROR_CHECK(err_code);
    err_code = low_power_pwm_duty_set(&low_power_pwm_1, 0);
    APP_ERROR_CHECK(err_code);
}
```

Inside the NUS handler, we have added code to control the two PWM channels based on the new duty cycle. Whenever the duty cycle changes, the NUS handler determines whether it is a hot (positive) or cold (negative) duty cycle, then turns the corresponding PWM channel on and the other off. It is worth noting that the opposite PWM channel must be turned off before the desired channel is turned on to prevent sending current in both directions at once.

The portion of the NUS handler that deals with PWM is given below.

```
if (duty_cycle > 0) {
       err_code = low_power_pwm_stop((&low_power_pwm_1));
       APP_ERROR_CHECK(err_code);
       err_code = low_power_pwm_start((&low_power_pwm_0), HOT_PIN);
       APP_ERROR_CHECK(err_code);
       nrf_drv_gpiote_out_set(RED_PIN);
       nrf_drv_gpiote_out_clear(GREEN_PIN);
       nrf_drv_gpiote_out_clear(BLUE_PIN);
} else if (duty_cycle < 0) {
       err_code = low_power_pwm_stop((&low_power_pwm_0));
       APP_ERROR_CHECK(err_code);
       err_code = low_power_pwm_start((&low_power_pwm_1), COLD_PIN);
       APP_ERROR_CHECK(err_code);
       nrf_drv_gpiote_out_clear(RED_PIN);
       nrf_drv_gpiote_out_clear(GREEN_PIN);
       nrf_drv_gpiote_out_set(BLUE_PIN);
} else if (duty_cycle == 0) {
       nrf_drv_gpiote_out_clear(RED_PIN);
       nrf_drv_gpiote_out_set(GREEN_PIN);
       nrf_drv_gpiote_out_clear(BLUE_PIN);
} else {
       // do nothing
}
```

Once the correct PWM channel is on, the PWM handler sets the duty cycle to the one calculated by the NUS handler and continues generating our PWM signal. Our PWM handler looks like this:

```
static void pwm_handler(void * p_context)
{
    uint8_t duty_cycle_converted;
    static uint16_t ticks;
    uint32_t err_code;
    UNUSED_PARAMETER(p_context);

    low_power_pwm_t * pwm_instance = (low_power_pwm_t*)p_context;

    if (++ticks > TICKS_BEFORE_CHANGE) {
        duty_cycle_converted = pwm_instance->period * abs(duty_cycle) / 100;
        err_code = low_power_pwm_duty_set(pwm_instance, duty_cycle_converted);
        ticks = 0;
        APP_ERROR_CHECK(err_code);
    }
}
```

In addition to using the Bluetooth connection to control the Peltier module, our device is also hardware capable of controlling it using physical buttons. To set this up, we would configure pins P0_7 and P0_8 as input pins, trigger an interrupt when one of these buttons is pressed, and control the PWM duty cycle within that interrupt.

Reading the temperature

Getting readings from the temperature sensor requires two main tasks: setting up the successive approximation analog-to-digital converter (SAADC) and setting up a timer to trigger it.

Our program uses one of the nRF52832's onboard real-time counter (RTC) modules to schedule and execute sampling. The nRF52832 has two timing options: the timer modules and the RTC modules. The key difference between these two options is their clocking mechanisms: the timers use the high-frequency clock; the RTC uses the low-frequency clock. The high-frequency clock draws 5-70µA on average, while the low-frequency clock draws 0.1µA on average. Since we are aiming to minimize power consumption and do not need to retrieve samples often, we would like to leave the high-frequency clock disabled and use only the low-frequency clock. Thus, we choose the RTC to schedule our temperature samples.

Of the three RTC modules available on our chip—RTC0, RTC1, and RTC2—the only one available for us to use is RTC2. Since our SoftDevice uses RTC0 and our BLE connection uses RTC1, we must choose RTC2 for the SAADC; tampering with one of the other modules renders our device unusable. The RTC2 module is configured with default settings. We then choose its prescaler value, which determines the interval at which it will trigger interrupts (32767 sets it to 1 Hz). Finally, we initialize it and set it to compare mode.

```
void rtc_init(void)
{
       uint32_t err_code;

       nrf_drv_rtc_config_t rtc_cfg = NRF_DRV_RTC_DEFAULT_CONFIG;
       rtc_cfg.prescaler = 32767;
       err_code = nrf_drv_rtc_init(&RTC2, &rtc_cfg, rtc2_int_handler);
       APP_ERROR_CHECK(err_code);

       err_code = nrf_drv_rtc_cc_set(&RTC2, 0, TIMER_PERIOD, true);
       APP_ERROR_CHECK(err_code);
}
```

Whenever the RTC counts to its prescaler value, it triggers an interrupt. Once the interrupt handler has been triggered and completed its function, we must clear the RTC's counter for it to count back up to the prescaler again.

To operate the device with a temperature sensor, the only further task would be to configure the SAADC. This would be completed as follows. By default, the nRF52832's SAADC is set to use an internal reference of 0.6 V and a gain of 1/6. Voltage range for the SAADC is calculated using the equation below.

$$input\ range\ = \frac{\pm 0.6\ V\ or \pm V_{DD}/4}{gain}$$

Dividing by our gain of 1/6 gives us an input range that reaches 3.6 V, which matches the voltage range output by the temperature sensor's voltage divider circuit. Thus, we would keep our gain and reference settings at their default.

The SAADC would operate in single-ended mode since no differential measurement is needed, and it would operate in one-shot mode since only one channel is being sampled. The SAADC also can be set up to oversample—that is, to take multiple samples over time and average them together to increase accuracy. In the nRF52832, oversampling should not be used when the SAADC is sampling on more than one channel, as this will cause it to average together samples from different channels. In order to preserve the possibility of adding more channels to the SAADC at a later date—for example, to sense battery level—we would disable oversampling. Finally, we would also choose to operate the SAADC in its low power mode state. We would trigger a sample in the RTC interrupt handler.

Operating the status LED

We have three modes of operation for our status LED: red, green, and blue. We can choose which mode the LED is in based on our duty cycle, as shown in Table 9:

Device state	**Duty cycle**	**Mode**
On (idle)	= 0	Blink green
Heating	> 0	Blink red
Cooling	< 0	Blink blue

Table 9. Status LED modes of operation

To blink the status LED, we can utilize the same instance of the RTC we already use for the SAADC. In the RTC interrupt handler, we turn the correct color on based on the device's duty cycle. After a delay, we turn it back off. The portion of our RTC interrupt handler that controls our status LED, then, looks like this:

```
void rtc2_int_handler(nrf_drv_rtc_int_type_t int_type)
{
        switch(int_type)
        {
                case NRF_DRV_RTC_INT_COMPARE0:
                        if (duty_cycle >  0) {nrf_drv_gpiote_out_set(RED_PIN);  } else
                        if (duty_cycle <  0) {nrf_drv_gpiote_out_set(BLUE_PIN); } else
                        if (duty_cycle == 0) {nrf_drv_gpiote_out_set(GREEN_PIN);}
                        else {/* do nothing */}
                        nrf_delay_ms(STATUS_LED_BLINK_WIDTH);
                        nrf_drv_gpiote_out_clear(RED_PIN);
                        nrf_drv_gpiote_out_clear(GREEN_PIN);
                        nrf_drv_gpiote_out_clear(BLUE_PIN);
                        nrf_drv_rtc_counter_clear(&RTC2);
                        break;
                default:
                        // Do nothing.
                        break;
        }
}
```

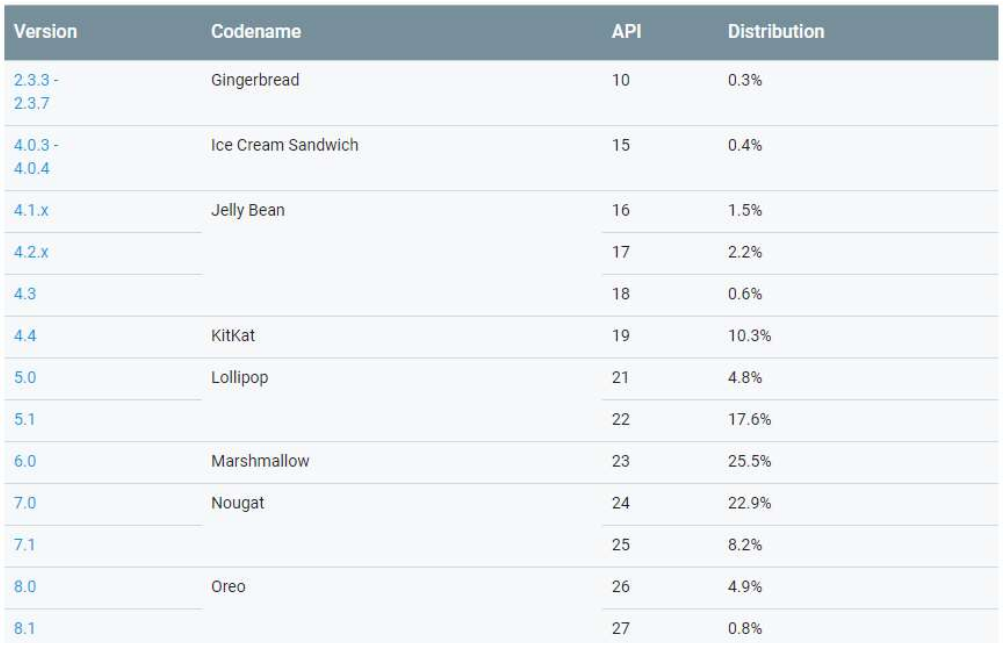
5. App

5.1. Introduction

This application was created for communicating with the microprocessor and to provide the user with a clean interface for controlling the temperature of the device and get useful feedback information from it.

General application information

Temperature Device Application is currently 7.31MB and designed strictly to run only Android phones. The minimum API (Application Programming Interface) version it supports is 23 (Android OS 6.0, Marshmallow) with a target API of 26 (Android OS 8.0, Oreo). According to Figure 13 and Figure 14 below, with our minimum API we can reach 62.3% of all Android users.



Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.5%
4.2.x		17	2.2%
4.3		18	0.6%
4.4	KitKat	19	10.3%
5.0	Lollipop	21	4.8%
5.1		22	17.6%
6.0	Marshmallow	23	25.5%
7.0	Nougat	24	22.9%
7.1		25	8.2%
8.0	Oreo	26	4.9%
8.1		27	0.8%

Figure 13. Android Chart API Distribution (Distribution Dashboard, Android Developers)

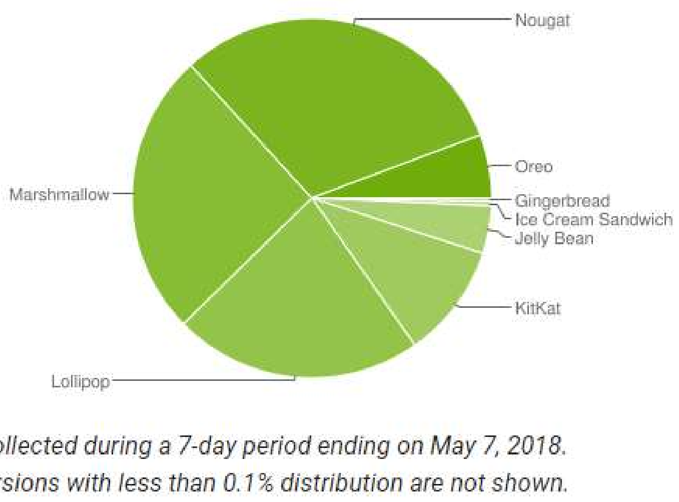


Figure 14. Android Pie API Distribution (Distribution Dashboard, Android Developers)

The application includes several non-compiler generated files. These files include one manifest, seventeen java classes, and thirty-one resource files (six drawable, fifteen layout, two menu, six raw, two values). These will all be referenced in detail in the sections below.

5.2. App flow

The general overview flow of the application can be split up into two main sections: the user interface flow (what is often considered the frontend) and the programmer interface flow (the backend). We will follow the logical flow for each. More detail for the frontend and backend which includes specific special cases that occur simultaneously in different threads will further be explored in section *User Interface* and section *Programmer Interface*.

User interface flow (frontend)

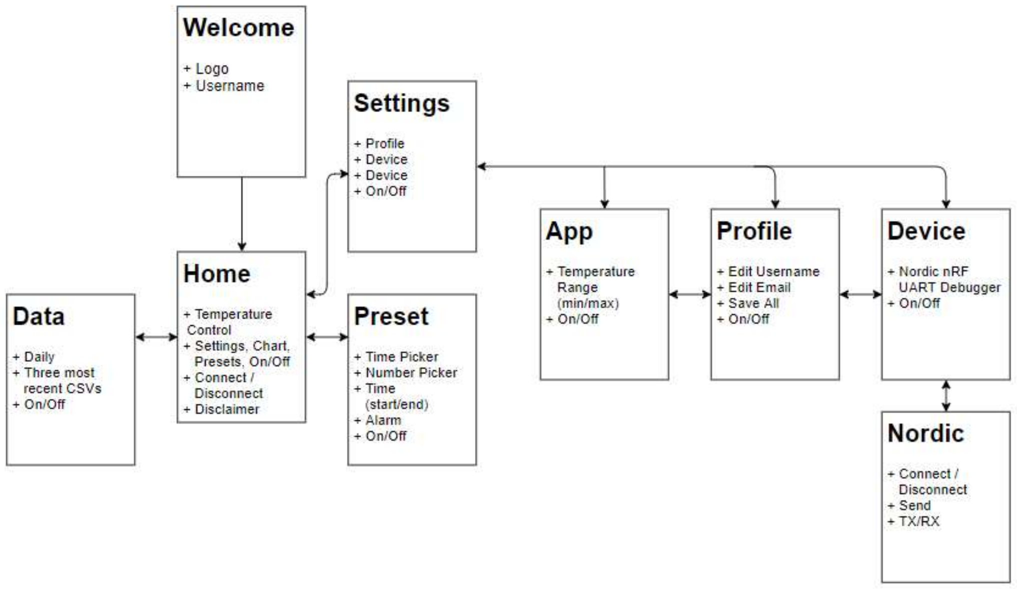


Figure 15. User interface flow chart

Following Figure 15 above, when the application is opened the first time, the first thing that is shown is a splash *Welcome* screen with the Temperature Device logo and text welcoming the user (username set in app). We proceed to the *Home* screen where the user has many options for new places to navigate to. Pressing the image buttons for the *Data* and *Preset* pages at the bottom left and right corners respectively lead to dead ends path wise but to several features for showing data collected and setting future temperatures. These two pages are child activities of *Home* which means we can easily navigate to *Home* with the press of the back button. A third child page is seen on the top right corner, a *Settings* image button. This *Settings* page is split into three sections called fragments. The first, *App Fragment,* allows the user to edit their app preferences. The second, *Profile Fragment,* allows for editing user information. The third, *Device Fragment*, is designated more for the programmer to debug the device. It provides useful feedback for the microcontroller programmer and keeps track of the external temperature device's current battery status.

Programmer interface flow (backend)

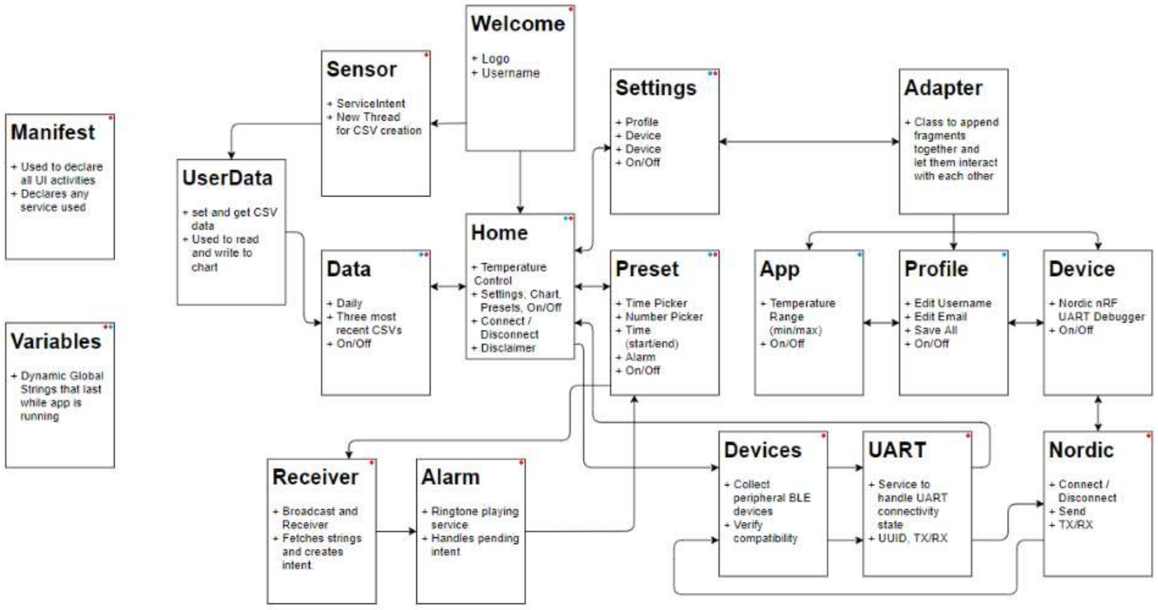


Figure 16. Programmer interface flow chart

Likewise, following Figure 16 above, the programmer interface flow begins with the splash *Welcome* screen and after leaving it we immediately begin a sensor thread used to collect data whenever it is appropriate and start the *Home* activity. The sensor thread often references the *UserData* class to update data which is later sent to the *DataChart* activity and output on a graph. From *Home*, like before, we can access our three pages *DataChart, Presets,* and *Settings* pages. The *Home* page uses Devices (*DeviceListActivity*) to find bluetooth devices around the phone and a UART service (*UartService*) to set up communication with our UART external temperature device.
From the *Home* page we can go to the *Presets* page. In *Presets* we call a Receiver class (*PresetsReciever*) once it has a preset ready to be delivered in the future. Once the preset's time comes it trigger an Alarm service (*RingtonePlayingService*) to start an alarm and set the chosen temperature for the device.
From the *Home* page we can also go to the *Settings* page in *Settings* we need to access an Adapter (*SettingsPageAdapter*) to populate *Settings* with three *App, Profile,* and *Device* fragments. The last *Device* fragment links to *Nordic*'s debugger app with also uses Devices (*DeviceListActivity*) and a UART service (*UartService*) like the *Home* page above.

5.3. User interface

Included in this section we explore the many options given to the user in the various pages provided throughout the application. The theme of flow will be followed to navigate through the pages and we will briefly mention some programmer interface (backend) functionality to clear up user interface (frontend) functionality. Both sides of the application are so intertwined that often we will jump back and forth between them.

Welcome screen splash page

`WelcomeScreenActivity.java [activity_welcome_screen.xml]`

First seen upon opening the app this splash *Welcome Screen* last for two seconds and then will jump into the *Home* page. The 'Welcome' text is later followed by the user's chosen username in the *Profile Fragment* part of the *Settings* page. This text is saved and retrieved from a `SharedPreferences` internal memory database later discussed in section *Programmer Interface - Memory Classes, Elements, and Services.* Figure 17 shows what the user would see.

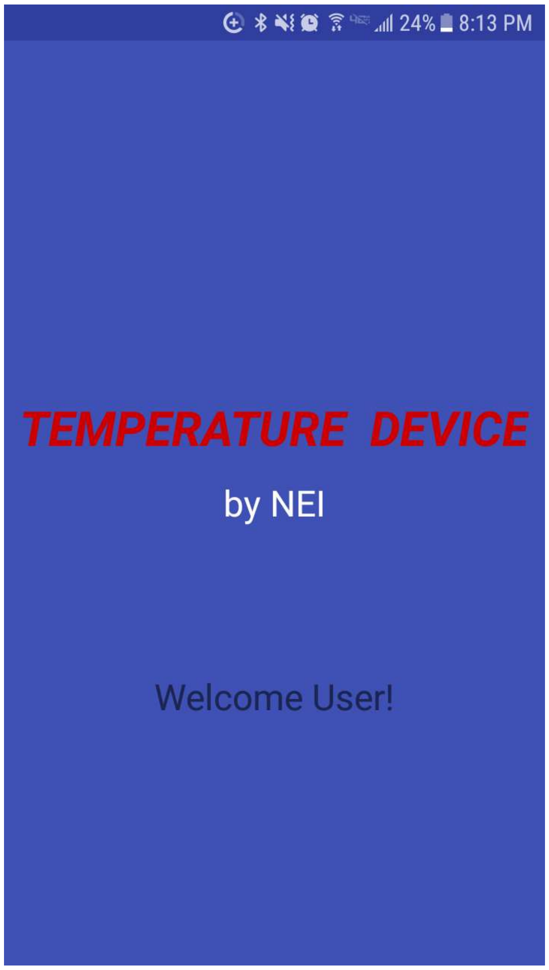


Figure 17. Welcome screen activity

Activities and their layout resources

`HomeActivity.java [activity_home.xml]`

If it is the first time the app is opened after being downloaded, then the first thing to show up on the *Home* page is a pop-up DISCLAIMER dialog box. This dialog box will not allow the user to continue unless they agree to the terms and conditions. If they disagree the app will force close. This DISCLAIMER dialog box is only ever preceded by another Bluetooth dialog box if the user has their Bluetooth turned off. This other Bluetooth dialog box asks the user to turn Bluetooth on and will not allow them to proceed until they do. This was added under the consideration that the entire app relies on Bluetooth connectivity. Finally, also upon start up, we begin our first secondary thread to collect data from the external temperature device's microcontroller. This thread will continue running in the background until the application is terminated. It will collect data only when allowed to by the user through granting a specific permission. This is further explored below in this section under *Data Chart Activity.*

Now that we have gotten passed the initial required *Home* permissions and settings we get to the main *Home page* seen below in Figure 18. The bottom left 'presentation of a line chart' ImageButton leads to the *Data Chart* page (upon being pressed) where the user can see their collected data for when the device was in use. The bottom right 'alarm clock' ImageButton leads to the *Presets* page (upon being pressed) where the user can set a preset for a future temperature. Finally, on the top left-most corner in the action bar of the page we see a 'gear' looking ImageButton that leads to the *Settings* page.

Next to the *Settings* ImageButton we have a lightened power ImageButton that doesn't lead anywhere but works as one of two safety power on/off buttons for the external temperature device (it can be visible on from any activity throughout the entire application). The second safety power on/off button is a physical on/off switch on the embedded board integrated in the temperature device. This lightened power ImageButton is its initial OFF state, once the user presses it it will change to a darkened power ImageButton representing the ON state. The memory for this is saved in dynamic memory later discussed in section *Programmer Interface - Memory Classes, Elements, and Services.* Unless the power ImageButton is in the ON state, the device will never output anything but a neutral neither hot or cold default '0' temperature.

In the center of the page we have the most important element, the temperature NumberPicker. This NumberPicker sets the temperature to be output and read by the external temperature device. The on/off button on the page mentioned above allows the spinner to move freely and select a temperature only if we are set to the ON state. If we are in the OFF state, the spinner will be locked, faded, and only show that it is set to '0' as seen below in Figure 18.

When the user is ready to use the device, they can press the CONNECT TO DEVICE button. This opens a dialog box that first asks the user to allow the application to access their location. The specific location permission being asked for is called `ACCESS_COARSE_LOCATION` and is needed to connect to surrounding Bluetooth device. Once the user agrees they will never be asked again but have the option to update permissions in their application settings on their phones. With permission granted a new dialog window appears showing the available devices (details on how devices are found in a new thread can be found in section *Programmer Interface - Bluetooth and Alarm Classes*). Once our device is found and clicked on the connection is ongoing until the user clicks the same CONNECT TO DEVICE button which now read DISCONNECT FROM DEVICE.

Text prompts in the form of TextView's can be seen below, to the left, and to the right of the CONNECT TO DEVICE button. Below the button *<Selcet a device>* will change according to the state of the device we are connecting to and then to the name of the device when we finally connect to one. To the right of *<Select a device>* and divided by a pipe '|' we see *RX:* which updates to show us exactly what we are receiving from the external temperature device's microcontroller. To the top left of the button we see a *Payload:* and a *TX:* TextView that both update to show what the current temperature the app is seeing (used for calculations and backend functionality) and what temperature is being transmitted to the microcontroller is respectively. Finally, to the top right of the button we see an *On/Off:* TextView showing us a Boolean value of the current state we are in (true for ON and false for OFF).

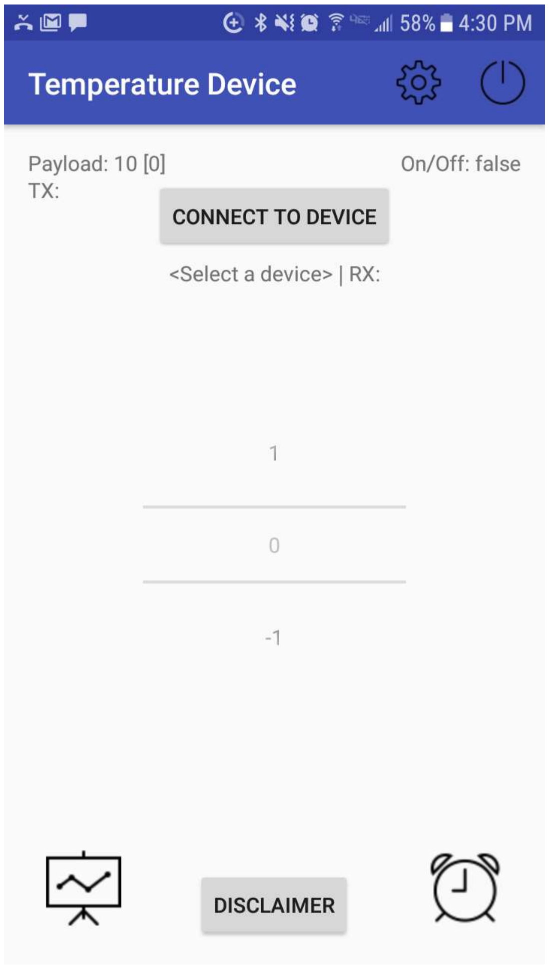


Figure 18. Home activity

DataChartActivity.java [activity_data_chart.xml]

From the *Home* page when the *Data Chart* ImageButton is pressed a permissions dialog box for *FILES_REQUEST* permission will appear. This permission is used to save collected temperature data onto a text file (.txt) located on the phones internal memory. Data is collected when the application is connected to a device and the power button is in the ON state. This data saved to the text file (automatically named daily to "TD:MM-DD-YYYY") is then collected from the text file and show on the graph Temperature vs. Time seen below in Figure 19.

On the bottom right, we see a spinner the user has the option of displaying one of the three most recent text files created and show their data on the graph. In this *Data Char Activity,* the user has the option of viewing the graph in portrait or landscape mode. Future versions will include an option to email yourself the selected text file at the push of a button.

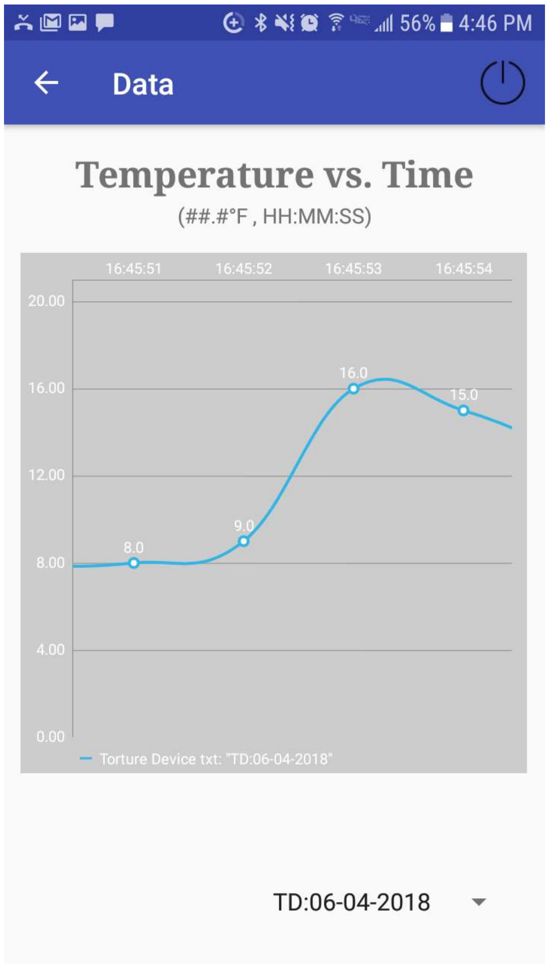
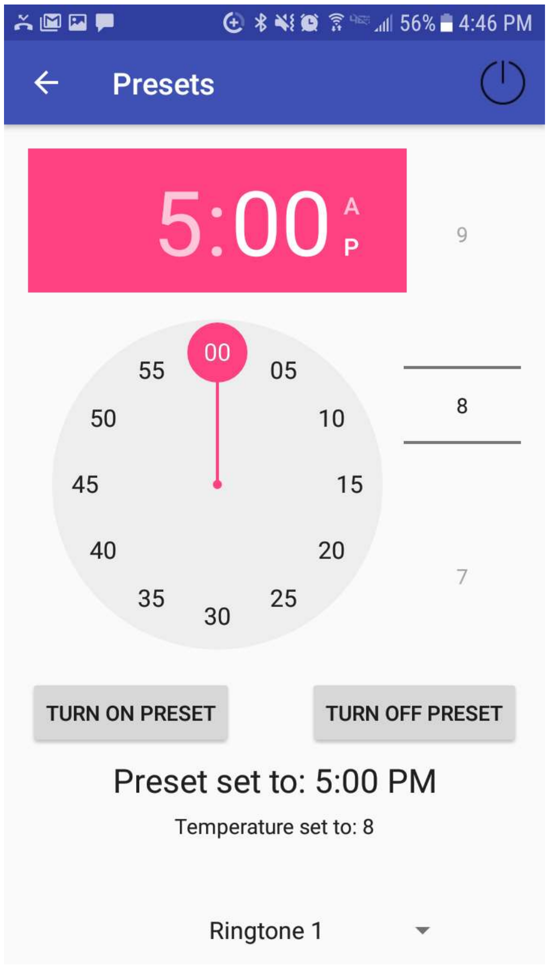


Figure 19. Data chart activity

PresetsActivity.java [activity_presets.xml]

On the *Presets* page we immediately are drawn to a large TimePicker on the left side of the page. This TimePicker defaults to the local current time when the page is opened. Setting the time is standard however the preset will not be set until the user presses the TURN ON PRESET button in Figure 20. When this occurs the TextView's "Did you set the preset?" text is updated to read "Preset set to HH:MM AM/PM". At this point whatever time is set along with chosen temperature from the NumberPicker on the right and alarm chosen from the spinner on the center bottom of the page will go off at the given time using a Pending Intent linked to a service routine (discussed in section *Programmer Interface - Bluetooth and Alarm Classes*). Once a preset is set the only way to turn it off is to press the TURN OFF PRESET button also seen in Figure 20. Once this button is pressed the TextView below it is updated to "Preset Off".

The default temperature is '0' and resets to '0' whenever the user leaves and returns to the page. This NumberPicker is also restricted by the users chosen temperature maximum and minimum discussed more in the next *Settings* section. The default alarm chosen is a silent mp3 file equivalent to no ring at all. The user has the option of choosing from two other mp3 files as well.



Preset set to: 5:00 PM

Temperature set to: 8

Figure 20. Presets activity

`SettingsActivity.java [activity_settings.xml]`

This *Settings* page is blank by default (seen in Figure 21) until we call fragments to populate it. A custom function iterates though known created fragments seen below and adds them in the order that they are called. The *Settings* page is then populated from left to right and a menu to select between the fragments added appears below the "Settings" page title. The user can also navigate between the fragments by swiping left and right in the *Settings* page.

Figure 21. Settings activity

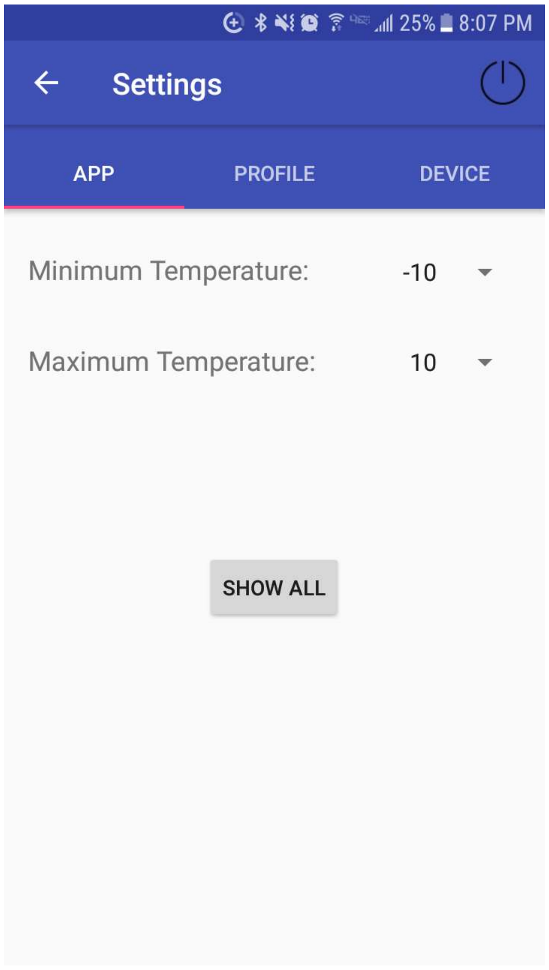
`SecctionsPageAdapter.java`

This java class is not connected to an xml page therefore its purpose is strictly to provide convenient functionally to the programmer. This class is used to create fragment objects. Each object takes in a title and then create a blank canvas that allows each fragment to add on their own xml styling page.

`AppFragment.java [app_fragment.xml]`

The *App Fragment* page is the first of three fragment pages set to the far left. Here the user can set the minimum and maximum temperature that they are most comfortable with. The temperature ranges for the minimum temperature are from –10 to 0 and 0 to 10 for maximum temperature. Once a number is picked it is saved automatically without the need of a SAVE button. The chosen temperature range dynamically updates the global temperature array through a function in the *Home* page. This array is then displayed on both the *Home* page's and *Presets* page's NumberPicker.

It is important to note that temperature '0' is inclusive in both ranges because we want to always be able to output a '0' temperature by default, when the user puts the ON/OFF button in the OFF state, and in case of emergency. The programmer can, in the background, transmit '0' to the microcontroller without the user's knowledge (in case of OFF state and in case of emergency) so '0' technically doesn't have to be inclusive (e.g. min-max can be 2-8). However, for a clean user interface, it was decided that '0' should be included so that the dynamic array can show '0' in the NumberPicker when '0' is being transmitted in the background.



Minimum Temperature: -10

Maximum Temperature: 10

Figure 22. App fragment

ProfileFragment.java [profile_fragment.xml]

The *Profile Fragment* page is the second of three fragment pages set in the middle. On this page the user can set their user and email. If the fields are empty (whenever the app is newly downloaded) there will be hints as seen below in Figure 23. These hints are updated to whatever the user updates their username and email to be. The username is used in the *Welcome Screen* to welcome the user and the email will be used in future implementations to send the user their data in the *Data Chart* page.

The SAVE (for username and email individually) and SAVE ALL (for both username and email collectively) buttons must be used to update the current static memory string values (mentioned in section *Programmer Interface - Memory Classes, Elements, and Services*) for username and email. If an empty string is entered, then a toast messages will alert the user of their empty submission and not save the empty string or overwrite a previously saved value.

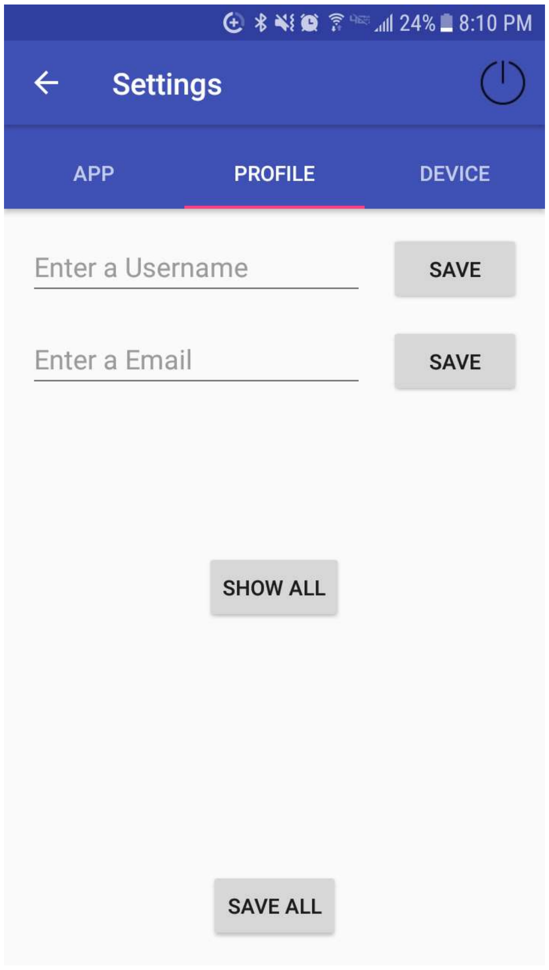


Figure 23. Profile Fragment

DeviceFragment.java [device_fragment.xml]

The *Device Fragment* page is the third of three fragment pages set to the far right. This page is used to help the microcontroller programmer debug their microprocessor code (seen in Figure 24 below). The center top NORDIC NRF UART button leads to Nodic's nRF UART open source debugger app mentioned in section *Extra Debbuger App: Extra Debug App.* In future implementations this page will also show the user an approximation of the temperature device's remaining battery life.

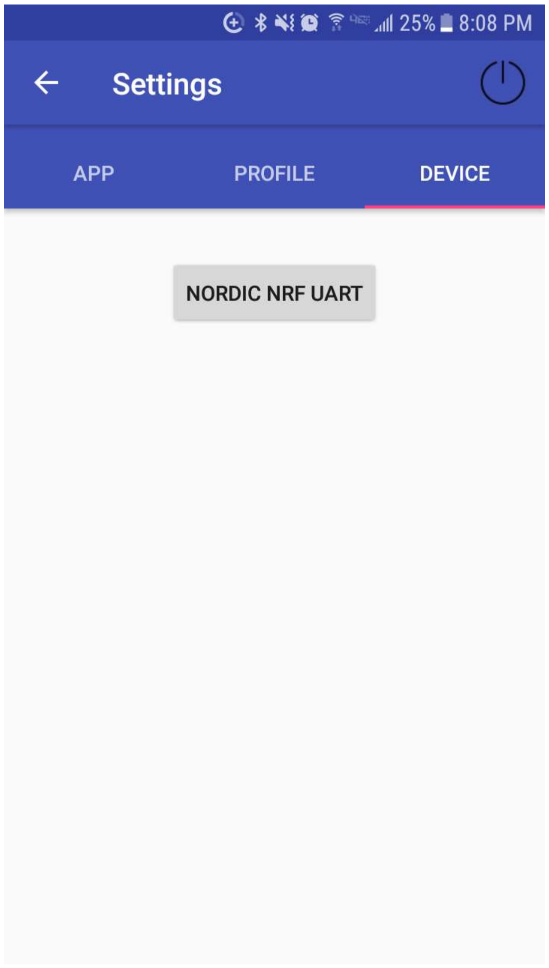


Figure 24. Device Fragment

Other resources

`Drawable:` `datachart.png` `presets.png` `settings.png` `poweroff.png`

`Poweroff.png` `nrfuart_hdpi_icon.png`

These drawable png's are used throughout the application as icons for ImageButtons, logo on *Welcome Splash Screen*, and logo for Nordic's debugging app. They are all located under Drawable which is under Android's res folder (resources folder).

`Menus:` `menu_all.xml` `menu_home.xml`

Two separate menus files are used. Menus are used to populate the action bar on whatever activity page they are called. The entire application needs access to the power button so the first xml, `menu_all.xml`, is used on all pages except on the *Home* page. On the *Home* page we need the power button and a *Settings* button to navigate to the *Settings* page. For this we use the second xml menu `menu_home.xml`.

`Raw:` `silence.mp3` `Katyperryhotncoldrintone.mp3`

`pokemonthemesongoriginal.mp3`

Raw files are downloaded along with the app. For the *Presets* page mentioned above we need these three files to be played along with whatever selection is made on the *Presets* page ringtone spinner.

`Values: Strings.xml`

Throughout the application specific strings values that are constantly being referenced are all stored in the `Strings.xml` file so that if they need to be edited they can be easily in one place. Another advantage to using this file instead of hard coding strings is that if the programmer ever wishes to translate the entire application they can do so easily with the push of a couple buttons.

`Phone Storage: data1.txt data2.txt data3.txt`

A maximum of three files will be stored on the phone to manage storage. The last three saved data files can be found under the phones internal memory storage under storage. Here the files can be renamed, altered, or deleted.

5.4. Programmer interface

The programmer interface, commonly referred to as the backend, is the magic behind the scenes that makes the whole application work. Everything here cannot be changed or ever viewed by the user.

Manifest

`AndroidManifest.xml`

The Android Manifest is used to declare all the activities used throughout the entire application. The way that these activities are linked to each other are also stated (parent and child relationships). All dependencies, external libraries, and permissions are also declared here before they can be used anywhere else in the code. Finally, all extra threads (that are also entire classes) are also declared here.

Memory classes, elements, and services

`Static Memory:`

Static memory is the memory that the application uses when it wants to store a value even beyond when the application is running. For example, if the application is force closed (a hard close where it is no longer is running in the background) then the memory we save in static memory stays and can be referenced the next time the application is opened. To accomplish this, we use something called `SharedPreferences` which is a simple, low memory using, key value pair database storage method. The way this works is we assign a key to a value we want to save. We retrieve the value by calling its given key. Core values that use static memory include: username, email, and min/max temperature.

`Dynamic Memory: GlobalDynamicStrings.java`

Dynamic memory is the memory that the application uses when it wants to store a value that will be used only when the application is running. When the application is closed the values will be lost and be set to their defaults when the application is started agian. For this type of memory, class *GlobalDynamicStrings* was used. This class used three main elements per string we want to save dynamically: a default value, getString(), and setString(). This class object works dynamically because it is constructed when the application is booted up and deconstructed upon its termination. Core values that use static memory include: OnOff, Payload, PayloadPreset, TempSet, and ReadyState.

`Text Files:` `UserData.java` `SensorServiceIntent.java`

To collect and save data to text files we need two java classes. The first class, *UserData*, is identical to *GlobalDynamicStrings* above. The reason for the extra class is to keep these dynamic memory variables separate from the rest used in the program to make life easier for the programmer. Core values that use static memory include: temperature, hms(hour:minute:second), and tx.

The second class, *SensorServiceIntent,* runs in an infinite loop from the time it's called (after the *Welcome* screen) until the application is shut down. It first checks to see if there exist a file for today's date that data can be written to. If there is then it locates that file and opens it, if a that file does not exist then it creates it. Using variables OnOff and Ready (from *UserData*) as arguments we check to see if we are ready to collect data. If we are then data is collected every second and written to the created text file that *Data Chart* activity can read from.

Bluetooth and alarm classes

`Bluetooth:` `HomeActivity.java` `DeviceListActivity.java` `UartService.java`

To have Bluetooth functionality on the *Home* page and in the debugger app (seen in section *Extra Debbuger App*) we use the *DeviceListActivity* class. This class is used to find local surrounding Bluetooth devices and then populates a list that the *Home* page used to prompt the user to pick a device. Once a device is chosen this same class checks to see if the device we want to connect is a UART device. This is done to further ensure that we only continue the pairing process with the correct type of temperature devices we need.

Once connection is achieved a *UartService* begins. This service solidifies the connection to our UART temperature device. Here we establish what UUID (Universally Unique Identifier) we are using, keep the connection ongoing/error out when connect is lost, read in characters, transmit out characters, bind and unbind to specific Bluetooth connections, establish GATT (Generic Attribute Profile – protocol for transfering BLE data), and update the user on the current state of the device (connected, disconnected, connecting, device not found, unable to connect, unspecified address). All the status information is passed on to the *Home* page and *Main* page (in the debugger application, section *Extra Debbuger App*) and then displayed to the user.

`Alarm:` `PresetsReciever` `RingtonePlayingService`

Once a preset is set in *Presets* we need *PresetReciever* that extends the built in Android BroadcastReceiver to bridge us to our *RingtonePlayingService.* Specifically, the *PresetReciever* passes preset state information and preset ringtone information to the *RingtonePlayingService* that it creates when the *Presets* page tells it to create it.

After the *Presets* tells the *PresetReciever to* create the *RingtonePlayingService* (occurring after TURN PRESET ON button is pressed in *Presets* page) we used the passed in data to do several things when prompted to. Creating the *RingtonePlayingService* allows us to travel down two avenues stated below.

Avenue one occurs if the time comes when the PendingIntent (something triggered to do something in the future) created in *Presets* goes off, then the selected ringtone and temperature that the user specified upon creating the preset is played and set. The PendingIntent turns the ON/OFF button to the ON state. Then the *Home* page is opened because that is where temperature control takes place (opening/jumping to *Home* occurs from within the app or away from the app if app is running in the background). And finally, a notification is created that will take the user to the *Home* page in case the user has their device in locked mode where applications cannot force themselves to the foreground.

Avenue two occurs when the user pressed TURN PRESET OFF button in the *Presets* page. If a preset is currently going off this button will turn off any ringtone alarm attached to it but not turn off the temperature. If the preset has not gone off yet it will cancel the PendingIntent and all the data saved in the *PresetReciever* and the *RingtonePlayingService.* Nothing will go off in the future.

5.5. Extra debugger app

To aid in the development and debugging of our embedded programmer for current and future development work an extra debugger app was added. This app was provided by Nordic Semiconductors and can be found on their git repository. It is open source free to modify as long as their commented header disclaimer remains in any copied or modified file.

nRF UART debugger

MainActivity.java

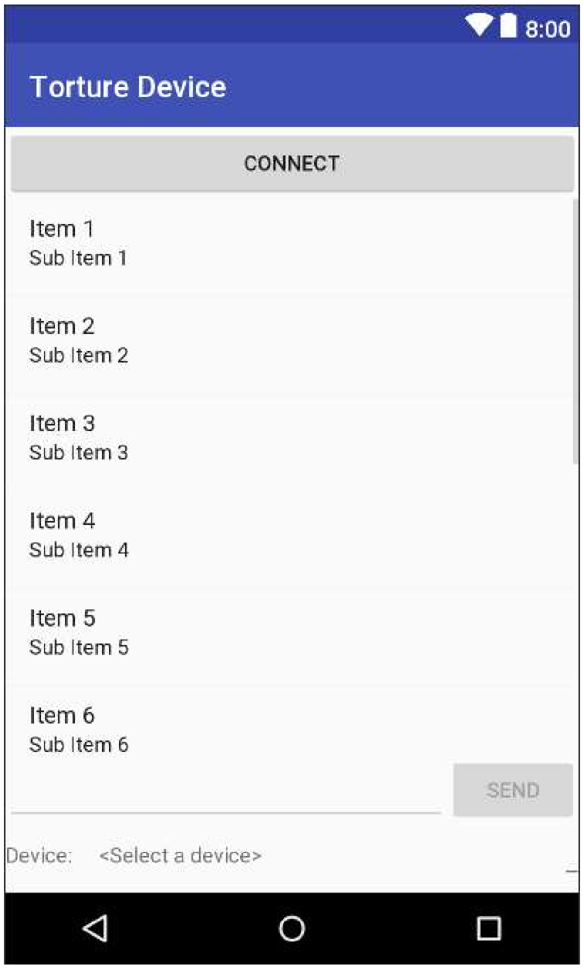


Figure 25. nRF Debugger

DeviceListActivity.java / UartService.java

[main.xml, device_element.xml, device_list.xml, dis_values.xml, message_detail.xml, title_bar.xml]

6. Testing

6.1. Placing the thermistor

Preliminary testing was done to determine the optimal location for the temperature sensor on the Peltier, determine the frequency to pulse the heat or cold at, and to determine the battery power. The testing was completed through a small program on an embedded board (NXP LPC11U24). The program regulated the frequency of the pulses through a variable resistor, outputted the value of the resistor, and outputted the temperature sensed from the temperature sensor.

The first task after completing the embedded board program was to test where the optimal location was to place the temperature sensor. Nine locations along the Peltier were tested by turning the pulse width of the power from 100% to 50%. Holding the temperature sensor against the Peltier, we were able to gather the results (Table 10) from each location (See Figure 26 for the locations).

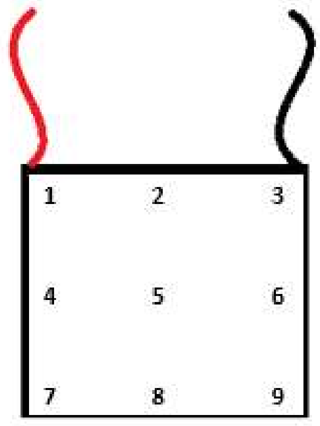


Figure 26. Testing locations for the thermistor

Testing Where To Place the Thermistor					
Location	**Pulse Width (%)**	**Voltage (V)**	**Location**	**Pulse Width (%)**	**Voltage (V)**
5	100	0.7	6	49.1	0.723
6	100	0.697	5	49.1	0.703
7	100	0.697	4	49.1	0.703
4	100	0.694	3	49.1	0.696
3	100	0.692	8	49.1	0.682
2	100	0.691	9	49.1	0.679
8	100	0.687	7	49.1	0.678
9	100	0.683	2	49.1	0.673
1	100	0.671	1	49.1	0.665

Table 10. Results of the thermistor location testing

From the results of the testing, it is possible to see that the temperature does not vary much between any of the locations. The most important location to sense is where it is hottest as that will affect the user greatest. Therefore, we chose to place the thermistor around location 5 and 6 on the diagram.

We attempted to attach the temperature sensor to the surface of the Peltier by hot gluing at first, and then by super gluing. Both methods did not work causing the temperature sensor to fall off. The most ideal method of attaching the sensor would be to create a case for the Peltier and heat sink that would hold the sensor against the Peltier.

6.2. Determining the pulse width

Each person reacts differently to temperature such as one user may tolerate heat better than a different user. It is pertinent to achieve one of our goals by making the wrist cuff user friendly. The user will be able to select one of the ten hot levels or ten cold levels on the app to increase the temperature intensity of the localized pulses.

To understand the range of temperatures people can handle, an experiment was created. Placing the Peltier on a volunteer's wrist, we were able to regulate the temperature by adjusting the power supply which was connected to the Peltier. With the data collected, we were able to create Figure 27.

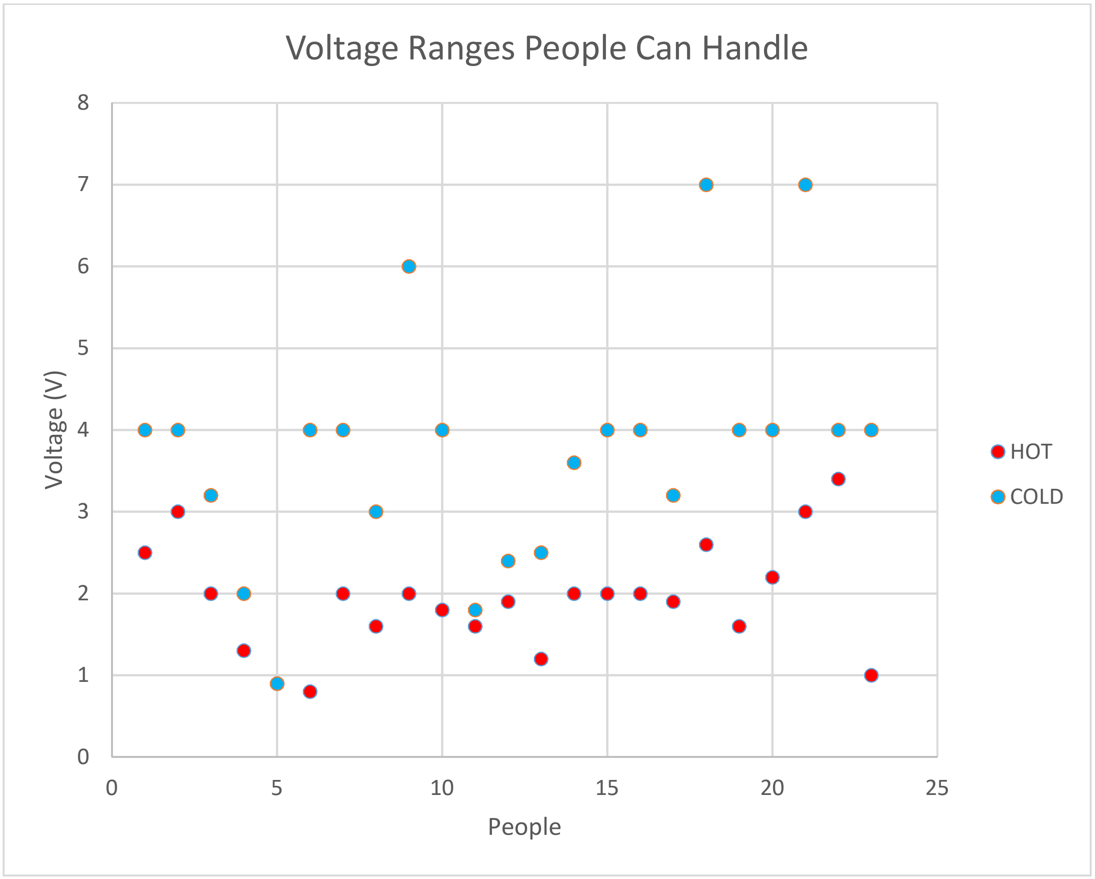


Figure 27. Volunteer test results

Since the microcontroller is limited to operate at a maximum of 3.7 volts, we limited our voltage to be at 3.7 volts, even though some volunteers did not feel uncomfortable until farther past this voltage. With this graph, we were able determine a limit on the temperature possible to output with the Peltier. To satisfy all the desired temperatures of the user, we decided to create a range of ten different levels for both hot and cold temperatures. Since the device will receive constant voltage from the battery, changing the duty cycle of the voltage entering the Peltier will allow us to control how hot or cold the device will get. Through experimentation, we found that the best way is to range the duty cycle from 100% to 0% in increments of 10%.

Another observation that was made during experimentation is that the user will become accustomed to the duty cycle. Therefore, we have decided to pulse the temperature at the desired duty cycle of the user for 45 seconds and then decrease the requested duty cycle by 50% for 15 seconds. For example, if the duty cycle is at 90%, it will run as so for 45 seconds and then decrease to 40% for 15 seconds preventing the user from becoming accustomed to the temperature related to the 90% duty cycle. If the user specifies the duty cycle to be 50% or lower, then during the 15 seconds the device will be off completely.

7. Results

7.1. Power analysis

After the prototype was created, some calculations were completed to determine the amount of power consumed by the entire system. The test involved measuring the amount of current that was consumed by the Peltier as the current consumed by the microcontroller is minimal. The results were surprising as we expected to be consuming 0.6 amps. Figure 28 shows that the current draw from the thermoelectric cuff is less than 0.6 amps allowing the time of usage to be good. From Figure 28 we were able to create a chart that describes the length of time the user can use the thermoelectric wrist cuff based on the level on the app they choose.

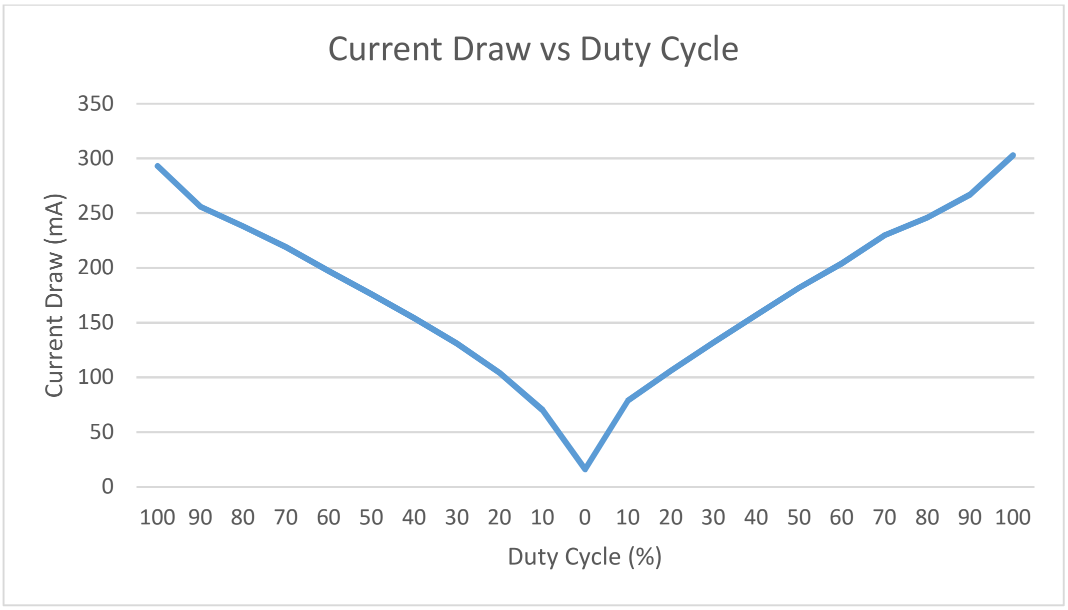


Figure 28. Current draw vs duty cycle

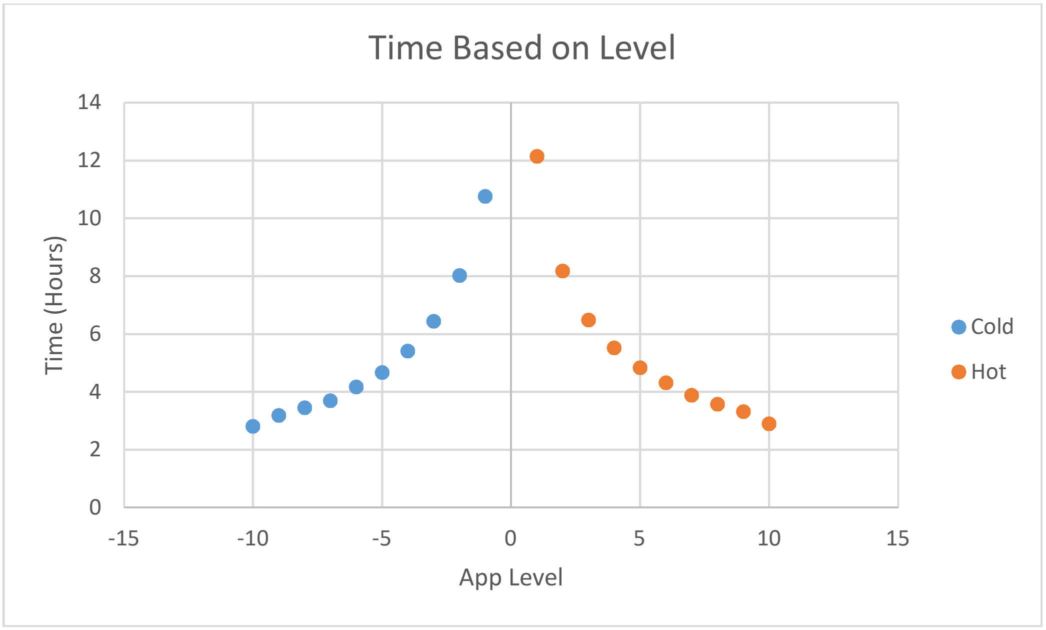


Figure 29. System current consumption measurements

From Figure 29, the calculations show that if the thermoelectric wrist cuff runs at the highest hot or cold setting, it can run for a little under three hours. If it were to run at the lowest hot or cold setting, it could run for around 12 hours before that battery would need to be recharged. There are several contributions to the conservation of energy. First, as discussed in the section above (*Determining the Pulse Width*), the temperature pulses last for 45 seconds at the selected setting and then drop five settings (or to completely off if originally on setting five or less). This not only conserves energy but prevents the user from becoming accustomed to the temperature pulses. Secondly, we have a status LED that indicates whether the device is on, pulsing hot, or pulsing cold. Rather than continuously having the LED on, it is pulsing every few seconds allowing some energy to be conserved while continuing to serve its purpose.

It is important to note that during the switching level one by one (ie. Level 2 to 3 on the app) requires an increase in current by 2mA during at most a three second period. The largest amount of power drawn during switching levels -10 to 10 (absolute hottest level to absolute coldest level) or vis-versa on the app is 460mA. Depending on how often and how many levels the user changes may decrease the amount of time the system can be used.

Also, to include in our results, all our goals for this project were met. First, the thermoelectric wrist cuff works wonderful as a therapy device. Natalie waited until she experienced symptoms from Raynaud's disease. This involved letting her fingers become mostly white indicating the lack of blood circulation flowing through them. She turned on the thermoelectric wrist cuff that was on her wrist and within less than five minutes color had returned to her fingers and she had full blood circulation in her hands. This was all done in a painless and effortless manner claiming the first goal of the project met.

The second goal was to create this device to be a personal temperature device. Other people who did not have Raynaud's disease used this device for a period of time and claimed that they felt more comfortable no matter the temperature. It was also user friendly enough to allow people to feel the freedom of being able to regulate their own temperature.

Our final goal was met through experimentation as well. If a person was going to their house for several minutes but did not want to turn on air conditioning or heating to use their house electricity, they could still feel more comfortable with the thermoelectric wrist cuff. This was proven with several volunteers. Overall our device was constructed well and as desired based on the goals we set in the beginning.

8. Parts cost analysis

Besides creating a user-friendly device, we also wanted to create it to be economical. In Table 11, we list each of the components with their price.

Part	Quantity	Price/Unit	Total Cost
Micro USB	1	$ 0.21	$ 0.21
Capacitor (4.7uF)	4	$ 0.10	$ 0.40
Capacitor (0.1uF)	1	$ 0.10	$ 0.10
Resistor (2k)	3	$ 0.10	$ 0.30
LiPo Charge Controller	1	$ 0.58	$ 0.58
Charger LED	1	$ 0.25	$ 0.25
Resistor (10k)	6	$ 0.10	$ 0.60
3.7V 850mA Lipo Battery	1	$ 9.95	$ 9.95
Red Bear Lab BLE Nano Kit v2 Microcontroller	1	$ 29.95	$ 29.95
L9110	1	$ 1.99	$ 1.99
Thermoelectric Peltier	1	$ 19.71	$ 19.71
Heat Sink	1	$ 3.34	$ 3.34
Thermistor	1	$ 0.75	$ 0.75
RGB LED	1	$ 1.95	$ 1.95
PCB	1	$ 1.00	$ 1.00
Total			$ 71.08

Table 11. Prototype Component Cost

The cost analysis shown above does not consider the man hours put into creating the thermoelectric wrist cuff. According to Embr Labs, they are selling similar devices for $299 (Embr Labs 2018). Embr Labs' device has similar functions as our device with the exception that it cannot be controlled from a phone app. Rather Embr Labs' device is controlled using buttons located on the side of the device to control the temperature.

Our device is user friendly and both accessible to use through an app and manually with buttons on the side of the device. With much speculation, we believe that if we began to manufacture this device, it would be possible to sell it for $199. This would include cost of parts, man-hours, manufacturing, and enough money to make a profit.

9. Impact

9.1. Health

As discussed in *Results* section, the thermoelectric wrist cuff has a beneficial impact on those with Raynaud's disease. Discussing our project with professor Kari Firestone, we speculate that the thermoelectric wrist cuff will be useful for those who require heat and cold therapy. This is easily achievable by cycling through both hot and cold temperatures possibly reducing inflammation in muscles and joints (Firestone).

9.2. Environment

Because our device can successfully change the user's perceived body temperature, the user could theoretically decrease the amount of time home or business HVAC is used. By doing so, the user can save energy, which has a positive impact on the environment. Air conditioners alone use 6 percent of all electricity produced in the United States costing about $29 billion to homeowners every year (Energy).

Our device aims to reduce the use of air conditioning and heating for both homeowners and business owners. Rather than using a large amount of energy to heat or cool a room, the thermoelectric wrist cuff will provide comfort to a user directly. With some speculation, energy usage could possible decrease with the use of thermoelectric wrist cuffs.

9.3. Sustainability

To ensure that our work stays relevant and encourage further innovation we have decided to make all our work open source. This includes all schematics, Gerber files, microcontroller code, and Android code. Our hope is that individuals will build upon what we have created to meet their needs. We also want to inspire said individuals to take ownership of what they create and not worry about any restrictions we would set. This way even after we finish development our users will still be able to modify and maintain their own devices. As long as programmers are ethical with our code we believe that our decision benefits all parties involved. Code for the Android application can be found on https://github.com/Skyllama83.

9.4. Ethics

By enabling users to feel comfortable in a variety of ambient temperatures, and by remaining open source and easy to implement, our device aims to increase pleasure and decrease pain for as many people as possible. To ensure that it does this effectively, we have considered a number of key ethical principles in its development. First, we have put our device through sufficient testing to ensure that it actually performs the task that it promises to perform—heating and cooling the wrist according to a user's preference. After our testing, we are confident that it can do so. Second, since the purpose of our device relates to health and medicine, we have made an effort to abide by the bioethics principle of non-maleficence—to do no harm (McCormick 2013). We have done this by implementing a number of safety features in the device, such as a physical on-off switch and a user-determined maximum and minimum temperature. Finally, to use our app—and, by extension, our device—we have required the user to provide their informed consent through a disclaimer window in the app. By making users aware of the risks inherent in our device through this disclaimer, and by requiring consent to these risks before enabling the app, we ensure that users are told the truth about how the device may affect them and have the opportunity to make an autonomous decision.

Resources

"Android Global Variable." Stack Overflow. Accessed April 28, 2018. https://stackoverflow.com/questions/1944656/android-global-variable.

Anthony, S. (2013, October 31). Wristify: A personal Peltier wrist cooler that could save the US millions in energy costs. Retrieved from https://www.extremetech.com/extreme/169951-wristify-a-personal-Peltier-wrist-cooler-that-could-save-the-us-millions-in-energy-costs.

ATTACTION. "Android Studio Tutorial - SWIPE SCREEN to Open Another Activity in Android. Hidden Touch Trick 2017." YouTube. June 06, 2016. Accessed April 17, 2018. https://www.youtube.com/watch?v=Q5Ndr944U2o.

"BLE Nano V2 (No Header)." RedBear Store. Accessed June 15, 2018. https://redbear.cc/product/ble-nano-2.html.

"Brief History of Thermoelectrics." History of Thermoelectrics. Accessed April 22, 2018. http://www.thermoelectrics.caltech.edu/thermoelectrics/history.html.

"Build Your Product." Bluetooth Technology Website. Accessed June 15, 2018. https://www.bluetooth.com/develop-with-bluetooth/build.

"Can Anybody Explain What Is Difference between Unbound and Bound Service in Android." Stack Overflow. Accessed May 7, 2018. https://stackoverflow.com/questions/25240299/can-anybody-explain-what-is-difference-between-unbound-and-bound-service-in-andr.

CodingWithMitch. "Android Tab Tutorial [Android Studio Tab Fragments]." YouTube. February 28, 2017. Accessed April 19, 2018. https://www.youtube.com/watch?v=bNpWGI_hGGg.

"Changing a Icon in ActionBar Depending on a Condition." Stack Overflow. Accessed April 29, 2018. https://stackoverflow.com/questions/11856440/changing-a-icon-in-actionbar-depending-on-a-condition.

"Distribution Dashboard | Android Developers." Android Developers. Accessed May 4, 2018. https://developer.android.com/about/dashboards/.

Domanico, A. (2014, October 09). Regulate your body's temperature with this wearable bracelet. Retrieved from https://www.cnet.com/news/regulate-your-bodys-temperature-with-this-wearable-bracelet/.

DrBFraser. "Read CSV Resource File: Android Programming." YouTube. February 26, 2017. Accessed May 8, 2018. https://www.youtube.com/watch?v=i-TqNzUryn8.

Firestone, K. Ph.D. (2018, January). [Personal]
Gyorki, John R., and Anonymous. "How to Select a Suitable Heat Sink." Design World. November 17, 2017. Accessed April 18, 2018. https://www.designworldonline.com/How-to-Select-a-Suitable-Heat-Sink/?cn-reloaded=1.

Hartle, Robert. "How Heat Sinks Work." HowStuffWorks. August 31, 2010. Accessed May 9, 2018. http://computer.howstuffworks.com/heat-sink1.htm.

"How to Use OnResume()?" Stack Overflow. Accessed May 7, 2018. https://stackoverflow.com/questions/15658687/how-to-use-onresume.

John's Android Studio Tutorials. "2/4 How to Add Buttons to Action Bar Android Studio." YouTube. July 27, 2015. Accessed April 29, 2018. https://www.youtube.com/watch?v=5MSKuVO2hV4.

Krull, L. PHD (2018, January). [Phone]

McCormick, Thomas R. "Principles of Bioethics." University of Washington School of Medicine. October 1, 2013. Accessed June 10, 2018. https://depts.washington.edu/bioethx/tools/princpl.html.

NASA. Accessed June 4, 2018. https://www.grc.nasa.gov/www/k-12/airplane/thermo.html.

"Nano2 Pinout." GitHub. Accessed June 15, 2018. https://github.com/redbear/nRF5x/blob/master/nRF52832/docs/images/Nano2/Nano2_Pinout.png.

"NordicPlayground/nRF52-ADC-examples." GitHub. Accessed May 20, 2018. https://github.com/NordicPlayground/nRF52-ADC-examples/blob/master/saadc_low_power/main.c.

Nordic Semiconductor Infocenter. Accessed April 12, 2018. http://infocenter.nordicsemi.com/.

"nRF5x/nRF52832." GitHub. Accessed June 15, 2018. https://github.com/redbear/nRF5x/tree/master/nRF52832.

"nRF52832." Nordic Semiconductor. Accessed June 15, 2018. https://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF52832.

Perez, Ito A. "Skyllama83." GitHub. May 15, 2018. Accessed June 15, 2018. https://github.com/Skyllama83.

"Personal Heating and Cooling Bracelet." AVANI SPINOFF.COM. Accessed May 26, 2018. https://spinoff.com/wristify.

Perry, J. Steven. "Add Dependencies To Android Studio." YouTube. July 04, 2016. Accessed May 6, 2018. https://www.youtube.com/watch?time_continue=287&v=4TsfgpGtBh0.

Radeff, Tihomir, and Yohohoasakura. "Save Text File to Storage in Android Studio." YouTube. November 30, 2017. Accessed May 10, 2018. https://www.youtube.com/watch?v=BnYruBLqdmM.

"Radio Versions." Bluetooth Technology Website. Accessed June 15, 2018. https://www.bluetooth.com/bluetooth-technology/radio-versions.

Rossi, Sergio. "AN4804 Application Note: How to Get Your Bluetooth Design FCC and BT Certified." January 2017. Accessed June 14, 2018. http://www.st.com/content/ccc/resource/technical/document/application_note/group0/f3/0c/31/34/bf/24/42/89/DM00257329/files/DM00257329.pdf/jcr:content/translations/en.DM00257329.pdf.

Saurel, Sylvain. "[Android] Learn How to Create a Real Time Line Graph with MPAndroidChart." YouTube. April 25, 2015. Accessed May 6, 2018. https://www.youtube.com/watch?v=a20EchSQgpw.

Smartherd. "#41 Android Tutorial : Selectors in Android - 1 - Make Your Android App - Part - 8." YouTube. June 30, 2014. Accessed April 29, 2018. https://www.youtube.com/watch?v=pjChIMkdKFo.

thenewboston. "Android App Development for Beginners - 62 - Custom Notifications." YouTube. January 14, 2015. Accessed April 23, 2018. https://www.youtube.com/watch?v=NgQzJ0s0XmM.

"Warm Up. Cool Down. Feel Good." Embr Labs. Accessed June 15, 2018. https://embrlabs.com/.

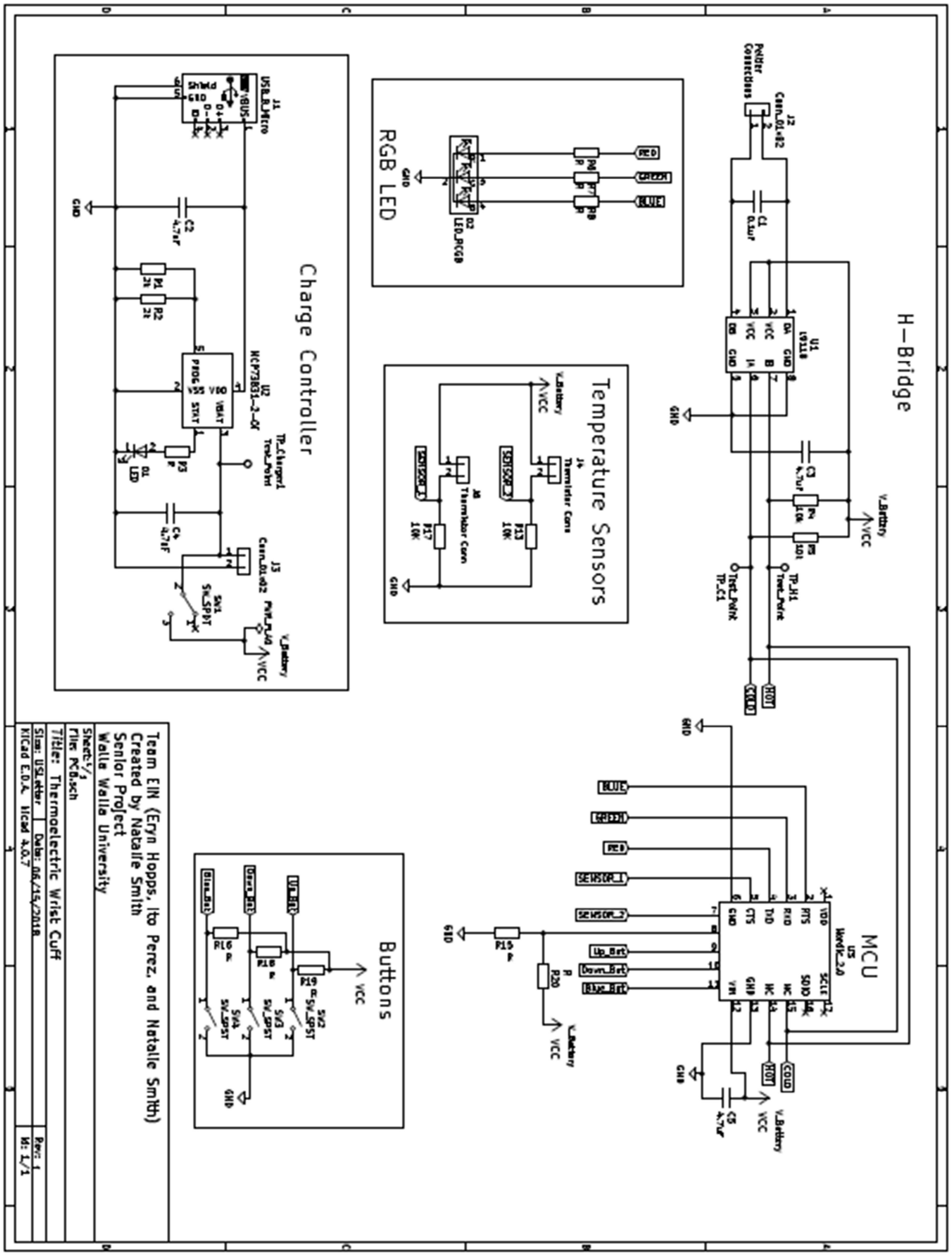
Xu, Anna. "Android Alarm Clock Tutorial: Part 1, Demo and User Interface." YouTube. January 18, 2016. Accessed April 18, 2018. https://www.youtube.com/watch?v=xbBlzOblD10&list=PL4uut9QecF3DLAacEoTctzeqTyvgzqYwA.

VS60. "Receiving Data from Mobile APP." Nordic DevZone. Accessed April 28, 2018. https://devzone.nordicsemi.com/f/nordic-q-a/23841/receiving-data-from-mobile-app.

"Zeolite." Wikipedia. June 13, 2018. Accessed June 15, 2018. https://en.wikipedia.org/wiki/Zeolite.

Appendix

Schematics


H-Bridge
Peltier Connections
J2 Conn_01x02
C1 0.1uF
U1 L9110
C3 4.7uF
V_Battery
VCC
R4 10k
R5 10k
TP_H1 Test_Point
TP_C1 Test_Point
HOT
COLD
GND
RGB LED
D2 LED_RCGB
RED
GREEN
BLUE
Temperature Sensors
J4 Thermistor Conn
J6 Thermistor Conn
SENSOR_1
SENSOR_2
R13 10K
R17 10K
Charge Controller
J1 USB_B_Micro
C2 4.7uF
R1 2k
R2 2k
U2 MCP73831-2-OT
TP_Charge1 Test_Point
D1 LED
R3
C4 4.7uF
J3 Conn_01x02
SW1 SW_SPST
PWR_FLAG
MCU
U5 Nordic_2.0
R15
R20
C5 4.7uF
Up_Bat
Down_Bat
Blue_Bat
Buttons
R16
R18
R19
SW2 SW_SPST
SW3 SW_SPST
SW4 SW_SPST
Team EIN (Eryn Hopps, Ito Perez, and Natalie Smith)
Created by Natalie Smith
Senior Project
Walla Walla University
Sheet: /
File: PCB.sch
Title: Thermoelectric Wrist Cuff
Size: USLetter
Date: 06/15/2018
Rev: 1
KiCad E.D.A. kicad 4.0.7
Id: 1/1