ACTIVE POSTURAL CORRECTION VEST

By

Jordyn Andrews
Aparna Srinivasan
Sophia Sulkar

May 2026

Abstract

The Active Postural Correction Vest (APCV) is aimed to be the solution of poor back posture through real-time physical reinforcement and reminders. The APCV is different than a normal physical therapy posture brace that usually rely on rigid support, which can lead to muscle atrophy if used too much. The APCV device utilizes stretch sensors sewn on to the backside of the vest to provide constant tracking of the user's back curvature. When a valid and significant slouch is detected, the main microcontroller sends a signal to the servo motors to tighten two cross-body back straps that are looped onto a spool. This provides a firm cue that encourages the user to fix their posture. To prevent false triggers from natural movement, the system requires the slouch to be sustained for 30 seconds before actuating. Once the user corrects their stance, the motors reverse to release the tension, ensuring the user actively engages their own core and back muscles. Additionally, the vest incorporates a hardware kill switch for immediate strap release in emergencies and pairs with a Bluetooth mobile application, allowing users to view live sensor data, adjust correction thresholds, and toggle between a dynamic Active mode and a static Brace mode.

Contents

1. Introduction 1
  1.1 Problem 1
  1.2 Solution 1
2. Outline of Subject Matter 2
  2.1 Introduction 2
  2.2 Subsystem Overview and Requirements 3
    2.2.1 Actuation and Power Subsystem 3
    2.2.2 Control Subsystem 4
    2.2.3 Sensing Subsystem 4
    2.2.4 Wireless/UI Subsystem 5
  2.3 Design 5
    2.3.1 Design procedure 5
    2.3.2 Power and Actuation Subsystem 5
    2.3.3 Sensing Subsystem 6
    2.3.4 Control Subsystem 7
    2.3.5 Wireless/UI Subsystem 7
  2.4 Design details 8
    2.4.1 Actuation and Power 8
    2.4.2 Sensing 10
    2.4.3 Control 11
    2.4.4 Wireless/UI 12
    2.4.5 Physical Design 14
  2.5 Verification 15
  2.6 Costs 19
  2.7 Conclusion 20
    2.7.1 Ethics 20
    2.7.2 Societal Impact 20
References 21
Appendix 22
  Full Schematics 22
  PCB Board Designs 24
  Flowcharts 27

Requirements and Verification Table 29

1. Introduction

1.1 Problem

Poor posture is a widespread and persistent problem, particularly among people who spend long periods sitting at desks for work or school. Hours of slouching and improper spinal alignment can gradually lead to musculoskeletal imbalances, chronic neck and back pain, shoulder strain, and even reduced breathing efficiency. Because these effects develop over time, poor posture is often ignored until it causes significant discomfort or long-term health complications.

Current solutions do not fully address the root of the problem. Traditional posture braces passively hold the body in place, but they often do all the work for the user, which prevents the user from actively engaging and strengthening the muscles needed to maintain proper posture independently. On the other hand, posture-monitoring devices that only send alerts or reminders rely entirely on the user to consciously correct themselves each time. This can be easy to ignore and does not provide physical assistance in forming better habits. Existing products either over-correct by removing user effort or under-support by offering no physical correction at all.

The core problem, therefore, is the lack of an active posture-correction solution that both assists the user physically and encourages their own muscular engagement.

1.2 Solution

We propose an Active Postural Correction Vest. Unlike passive braces, this system uses an active electromechanical feedback loop to physically retrain the user's posture, then letting go of the tension so that good posture is maintained by the user, not just the device itself.

The device itself consists of a wearable vest equipped with a stretch sensor placed along the middle of the user's back to monitor posture changes. When the user slouches, the conductive stretch sensor's resistance increases and shifts away from the calibrated upright-posture threshold. After the system detects a sustained slouch for 30 seconds, the Power & Actuation PCB triggers the servo motors mounted on the back of the vest. These motors reel in a ribbon-based cabling system connected to the shoulder straps, gently pulling the user's shoulders back into proper posture. Once the user returns to an upright position, the motors reverse and unspool the ribbon, releasing tension so the user can move naturally and maintain posture on their own. In terms of safety, the device is designed to apply gentle corrective tension rather than strong force. A kill switch is also included so that when the user activates it, the motors immediately unravel the ribbon and the system completely shuts off. The system also reduces false triggering by smoothing noisy sensor readings and using a 30-second delay so that brief natural movements, such as leaning or shifting, do not activate the motors.

The Bluetooth app allows the user to scan for and connect to the APCV device, view live posture data, monitor the stretch sensor value, see the current posture state, check whether the kill switch is active, and view the current mode and motor movement amount.

2. Outline of Subject Matter

2.1 Introduction

The overall function of our Active Postural Correction Vest has stayed consistent throughout the design and implementation process. We have created a vest that monitors posture using the same 4 subsystems as we stated in the proposal: Sensing, Control, UI/Wireless, and Actuation and Power (*shown in the Block Diagram in Figure 1*). Overall, this vest's function is to physically correct a user's posture using Servo motors, then release tension in the straps after the user has maintained their own good posture for a threshold of time so that the user learns to maintain good posture habits on their own.

In "Active Mode", which can be toggled in the wireless application, the straps start at a loose state, allowing the user to move around naturally. In this state, we have implemented a 30 second timeout for any slouch activity, as we only want the vest to actuate if slouch is consistent, not if there are small movements like bending down or reaching for something. After the user has maintained a consistent slouching posture for 30 seconds, the Servo motors actuate at the same time, reeling in the straps and forcing the user to get into a state of good posture. After 10 seconds of the user maintaining this good posture state, the Servo motors reverse and unravel the straps, returning to the original state where the straps are loose and the user can move freely again.

In "Brace Mode", which can also be toggled in the app, the straps are immediately reeled in to the "good posture" angle, where they stay until brace mode is either toggled off or the kill switch is pressed. This mode is meant for anyone who might need extra help maintaining good posture at first, with the goal being that eventually the user can start using the vest in Active Mode. This mode should teach the user where their "good posture" should be, and help them stay consistent with it until they feel that they are ready to maintain good posture on their own. This mode can also be used for anyone who has an injury or back problem where they need constant support to stay in good posture, as it holds the user's back in place.

One very important aspect of this vest is the kill switch. Since safety is our top priority, we have an easily accessible kill switch on the vest so that the user can stop all vest functions in case of failures or problems with the vest. When the kill switch is pressed, if the motors are in the "reeled-in" or "actuating" state, they will immediately reverse to their "loose" state so that the user can easily take off the vest. When the user is ready to turn the vest back on, all they need to do is press the kill switch again and all processes will start fresh.

2.2 Subsystem Overview and Requirements

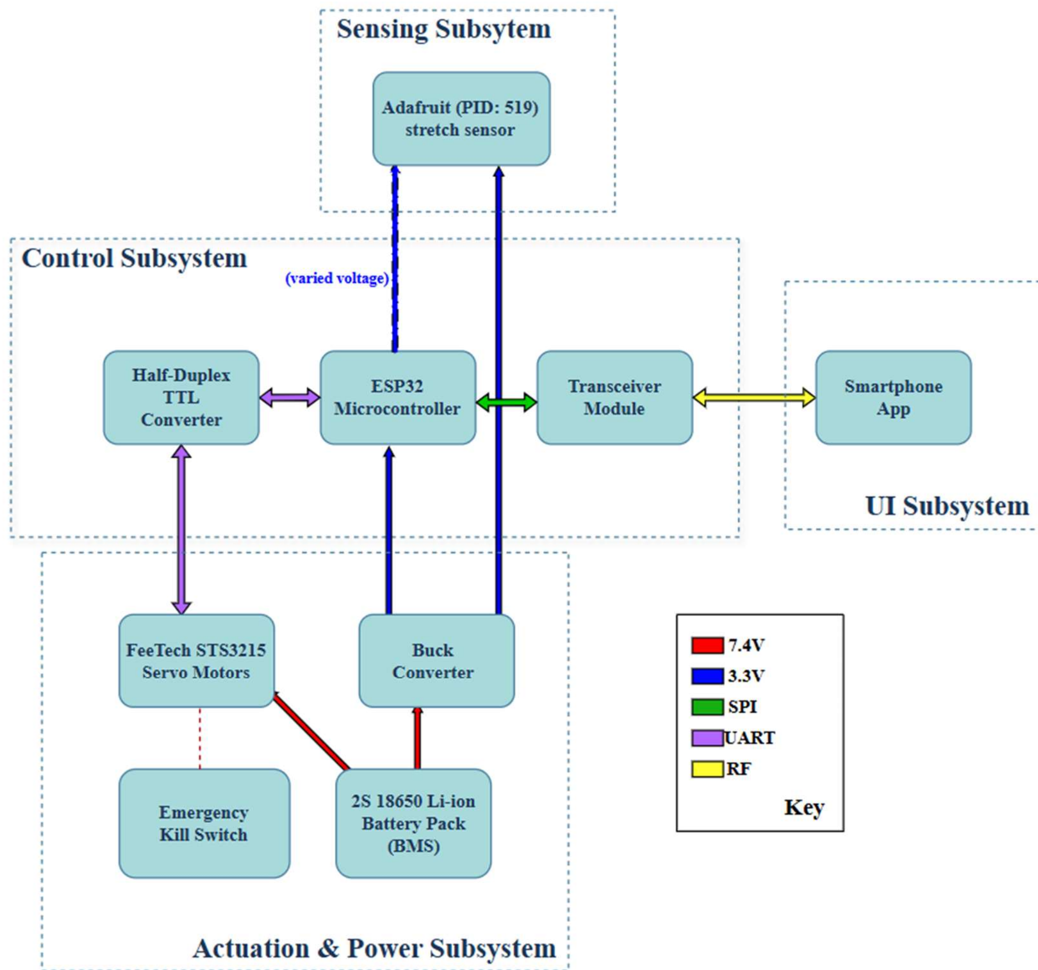

Figure 1: Block Diagram for Hardware/Electrical Design

2.2.1 Actuation and Power Subsystem

The actuation and power subsystem provides all the electrical power and drives the physical correction capabilities of the vest. This subsystem consists of the motors along with the battery pack and now buck converter, which was changed from our original design since we found it to be a much more reliable way of stepping down voltage in terms of less heat generation and better efficiency. Due to some issues in testing, we had to take out the battery pack and kill switch in this subsystem and used the DC Voltage Supply in the lab as the battery and wired up the kill switch in the Control Subsystem instead. The voltage regulator generates a stable 3.3V for the ESP32. This subsystem connects "upward" to the Control and Sensing subsystems via the 3.3V rail.

Subsystem Requirements:

1. **Servo-supply connection:** This component is necessary in order for the servos to generate the necessary tension to correct posture. If the motors are not supplied enough voltage or are

supplied with too much voltage, the system will completely fail, as the motors will not be able to move or will burn out.

2. **Voltage Regulation:** This must be fully working as it allows the microcontroller and sensors to run, as they expect the voltage to be stepped down to 3.3V. If this did not work, the microcontroller could completely be fried, as it cannot take the 7.4V we supply to the system.

2.2.2 Control Subsystem

The control subsystem executes posture detection as well as safety decisions. The ESP32 reads stretch sensor ADC values and smoothes them to prevent outliers from affecting the overall state, applies logic to confirm sustained slouching, and issues motor commands when correction is needed. In designing this subsystem, we also changed the kill switch to be controlled by the microcontroller in this subsystem rather than the Actuation and Power subsystem, as we wanted the motors to unravel before power was completely cut to allow for the user to take of the vest easily in a "loose" state. This subsystem connects to the Sensing subsystem via ADC inputs from the stretch sensors. It connects to the Actuation subsystem via a half-duplex UART servo bus. Lastly, it connects to the Wireless/UI subsystem via Bluetooth.

Subsystem Requirements:

1. **30 Second Delay Logic:** This is required for accuracy in slouch event detection, ensuring that the vest will not correct for temporary movements like bending over or stretching. If we did not implement this, there would be motor actuation every time the user moved out of a specific place, which is not the overall goal of this project, as we want to help users correct long-term slouching.
2. **TTL Converter:** This is vital for the microcontroller to communicate with the servo motors. Since the microcontroller is full-duplex and the motors are only half-duplex, there must be a conversion so that they can talk to each other.
3. **ESP32 Microcontroller:** This is necessary for data to be processed from the stretch sensors, ensuring safety and overall actuation. This is the core logic of this vest, and if it fails, essentially nothing can be done with the vest.
4. **E-Stop:** This has been changed from being a requirement in the Actuation and Power Subsystem to the control subsystem, as we wanted the motors to unravel to a loose state before the user took off the vest. This must be working to ensure user safety, as this is a wearable device.

2.2.3 Sensing Subsystem

The sensing subsystem provides posture measurement signals. In our updated design, there is one conductive rubber stretch cord mounted straight down the user's back, as this is the area of the back that provides the most accurate resistance readings pertaining to slouching. Each sensor is conditioned into an analog voltage through a voltage-divider circuit and read by the ESP32. In addition, servo load is treated as a safety measurement that can replace pressure sensors. This subsystem connects to the Control Subsystem through ADC (stretch).

Subsystem Requirements:

1. **Connection with ESP32:** The sensors must be connected to the ESP32, as if it was not, we would not get data from the stretch sensors. This is vital for setting slouch thresholds and triggering motor actuation to correct posture, so this is a non-negotiable requirement.
2. **Cord in good condition:** The conductive rubber cord must be in good condition, as this is the only way that we detect slouch for our system. If this fails, we will not be able to retrieve accurate data from the conductive cord, and posture state could get into an incorrect state, which would create issues in the functionality of the vest.

2.2.4 Wireless/UI Subsystem

The Wireless/UI Subsystem provides user interaction, calibration, and logging. A smartphone app exchanges BLE packets with the ESP32 to run calibration to the specific user (ie. setting the Motor Target angle for good posture), to visualize sensor readings and the corresponding posture state, to set the preferred mode (Brace or Active), and to see the kill switch status. This subsystem connects only to the Control subsystem (wireless data) and applies those parameters to sensor interpretation and actuation behavior.

Subsystem Requirements:

1. **Connection with ESP32:** It is vital to have this connection to the Microcontroller, as without it we would not be able to get the necessary data about sensors/motors, nor will we be able to calibrate the vest to each unique user.
2. **User-Friendly Interface:** Users must be able to easily navigate the app, as it holds all of the calibration and posture data, along with facilitating Bluetooth connection to the vest itself. If the UI is too difficult to figure out how to use, users will not be able to set any parameters for their brace and may have a hard time calibrating their vest or seeing their posture habits.

2.3 Design

2.3.1 Design procedure

2.3.2 Power and Actuation Subsystem

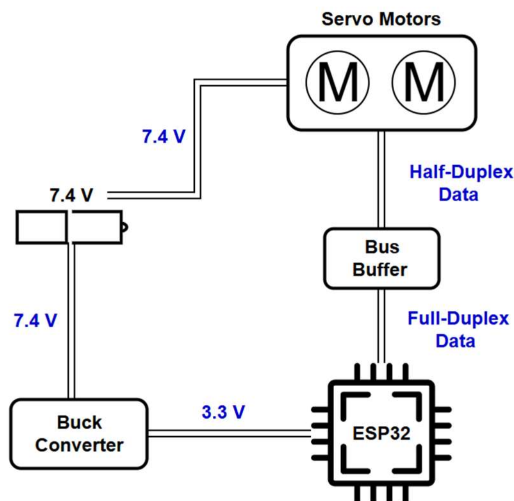

Figure 2: Basic Circuit Schematic of Power & Actuation Subsystem

The Actuation & Power subsystem at its core is made up of three main components; the power source, the motors, and the voltage/data conversion ICs, as seen in Figure 2. We selected the FeeTech STS3215 7.4 V servo motors as our actuators because they provide the necessary high torque while operating natively at 7.4 V. This eliminates the need for bulky, inefficient boost converters, allowing the system to draw the required 3 A peak current directly from the power supply. Powering the motors directly from the 7.4 V rail prevents current bottlenecks and ensures the motors do not stall during a corrective pull.

To power the ESP32 microcontroller and sensing logic, the 7.4 V supply must be stepped down to 3.3 V. We chose a switching buck converter over the initially proposed AZ1117 linear regulator. The buck converter prevents voltage sags during sudden motor current spikes, providing a highly stable 3.3 V rail that prevents the microcontroller from browning out or resetting. The ESP32 utilizes full-duplex UART, while the servos require a 1-wire, half-duplex TTL bus. To handle this conversion reliably, we implemented a hardware bus buffer. We specifically chose the SN74LVC126DBVR over a standard 74HC series buffer. The LVC logic family is optimized for 3.3 V operation and can safely interface with the servo's data line, whereas standard HC logic typically requires a 5 V supply to reliably meet the necessary switching speeds and input voltage thresholds.

2.3.3 Sensing Subsystem

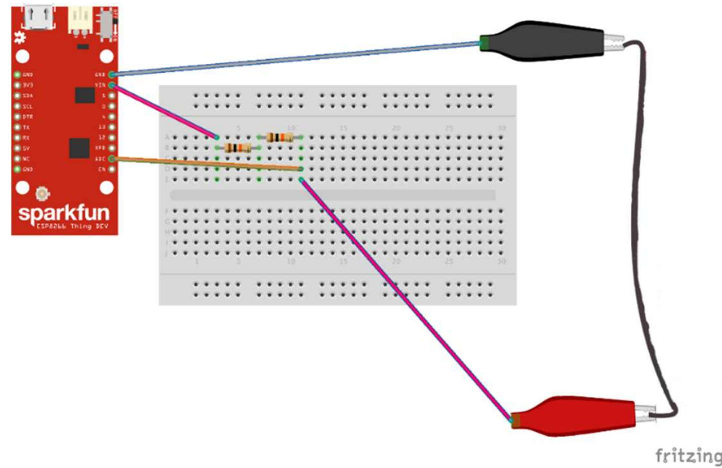

Figure 3: Stretch sensor circuit used in both our breadboard & our PCB [8]

We selected the Adafruit conductive rubber cord to act as our primary stretch sensor. During the design phase, we considered more complex alternatives, such as Inertial Measurement Units (IMUs) or arrays of rigid flex sensors. Our system's primary goal is to identify significant and prolonged slouches, so gathering highly granular spatial data is unnecessary. Measuring simple resistance changes via a stretch cord was the best option as it drastically reduces computational overhead while still providing a robust and reliable threshold for slouch detection. Because this method of posture detection proved to be highly reliable and computationally lightweight, the fundamental electrical design of the sensing subsystem was kept exactly as originally proposed. The electrical design just consisted of a simple

voltage divider circuit and probes on either end of the cord as seen in Figure 3. The only modification made during the physical prototyping phase was adjusting the placement of the cord on the vest to ensure it had the maximum possible stretch delta during a slouch event.

2.3.4 Control Subsystem

The Control Subsystem implements logic for the 30 second delay time when detecting a slouch event, interprets sensor data, commands motor actuation, and enforces the limits on safety for the vest.

This Subsystem includes the ESP32-C6-WROOM-1 Microcontroller, which requires 3.3V from the voltage regulator in the Actuation and Power Subsystem. This microcontroller has a built-in Transceiver module which will communicate with the microcontroller via SPI. This Transceiver Module will interface with the UI subsystem via RF, which will allow for Bluetooth connection to the mobile app. The BLE capability of the Microcontroller was what made us use it for our app. Alternatively, we could have chosen an alternate method of feeding data to our app, like WiFi, but Bluetooth was a better choice in this case due to the fact that the system we have does not require internet-connected, high-data communication, and Bluetooth made it much easier to pair a local device with a very simple and direct communication style.

The microcontroller also must interface with a Half-Duplex TTL converter in order to communicate with the Servo motors. This TTL converter is necessary because the ESP32 uses a standard 2-wire UART protocol with TX and RX buffers, while the STS3215 servo motors that we are using require a 1-wire half-duplex data line. Essentially, the UART communication between the ESP32 and the TTL converter is full-duplex, while the UART communication between the STS3215 servo motors and the TTL converters is half-duplex, which is why conversion is necessary here.

The ESP32 interfaces with the Adafruit stretch sensors in the Sensing Subsystem via ADC readings. This was our best choice of value to use for the posture detection, as these resistance values are accurate to when the user slouches, as the resistance values increase, and when the user has good posture, they decrease. In our original design, we proposed to use the converted voltage values as the threshold for slouching in our control code, but upon further investigation into threshold setting in the testing phase of our vest, using the resistance values with data smoothing yielded much more accurate sensing and threshold settings. The resistance values varied much more than the voltage values did when slouching and keeping good posture, as there was a difference of several thousand when the stretch sensors were stretched/relaxed, versus the voltage values which only ranged from 3.1V-3.3V, and did not change much outside of that, so they were not useful in slouch detection.

Lastly, the ESP32 communicates with the Kill Button on the front panel in order to stop all functionality of the vest in case of emergency. This is a safe way to return the motors to an unraveled state in order for the user to be able to easily take off the vest. Alternatively, we could have used our original idea of cutting all power to the vest completely through the Power and Actuation subsystem, but we found that it was actually much better to set this kill functionality in software rather than just cutting all power immediately through the Power and Actuation subsystem, as we could actually unravel the spools and return them to a loose state if they were already actuating so that the user could more easily take off the vest in an emergency event.

2.3.5 Wireless/UI Subsystem

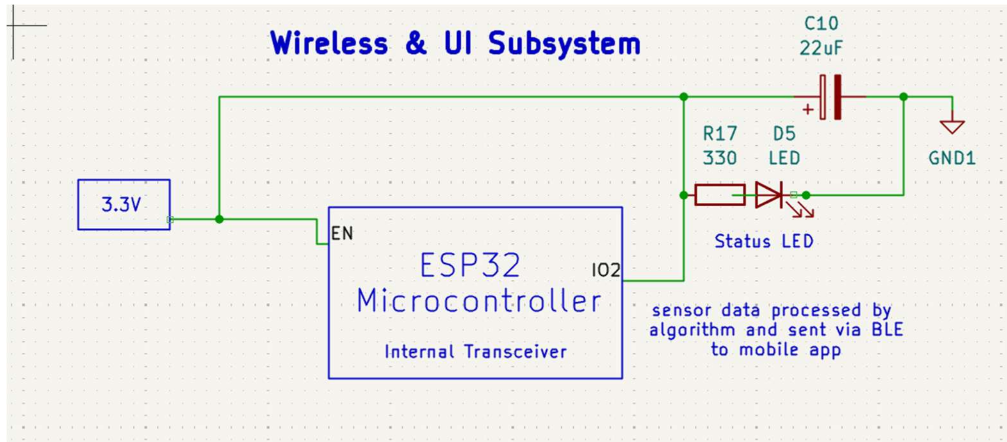

Figure 4: Basic Schematic of Wireless & UI Subsystem

The User Interface Subsystem overall makes posture state tracking and user-adjustable utilities available. As seen in Figure 4, the smartphone app communicates with the Transceiver Module in the ESP32 in the Control Subsystem via RF, or Bluetooth, and has a Connection Status to show the state of the Bluetooth connection. It gets data from the microcontroller for features like live posture tracking and sensor values, mode status and setting, Servo angle movement setting, kill switch activity, and slouch threshold viewing for the user.

We decided to create a Wireless App for this vest because we wanted to allow the user to actually be able to see their posture habits as they are using the vest. In the mode toggling section, we included the two main functionalities of the vest: Brace and Active. These modes were chosen because we wanted our vest to have multiple different functions to fit a wider scope of users. Active mode was more of an obvious choice to include, as it represents the main purpose of our vest: physically correct the user when they are slouching for a long time, but do not hold the user in a braced position. Brace mode was included in our design because we wanted to ensure that any user of this vest could choose to use our vest as more of a back brace in the case of an injury or just be held in good posture so that they can learn what position they need to be in to have good posture.

2.4 Design details

2.4.1 Actuation and Power

Power enters the system from the 7.4 V 2S Li-ion battery via an XT30 connector (J1). Because the Feetech STS3215 servo motors can draw large transient currents (up to 3 A at stall), robust protection and decoupling are required at the input stage.

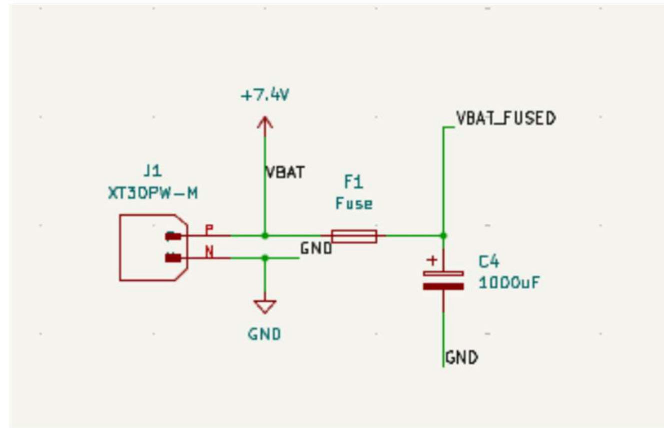

Figure 5: Battery & Fuse Sub-Schematic

As seen in Figure 5, The raw battery voltage (VBAT) is immediately routed through an inline fuse (F1) to protect the circuit and the user from short circuits or dangerous overcurrent events, resulting in the protected VBAT_FUSED net. To prevent severe voltage droop when the servos actuate and suddenly draw high current, a large 1000 µF bulk electrolytic capacitor (C4) is placed across the power rail. This bulk capacitance acts as a local energy reservoir, keeping the 7.4 V rail stable.

To power the ESP32 and the logic-level ICs, the 7.4 V VBAT_FUSED line must be stepped down to a clean 3.3 V.

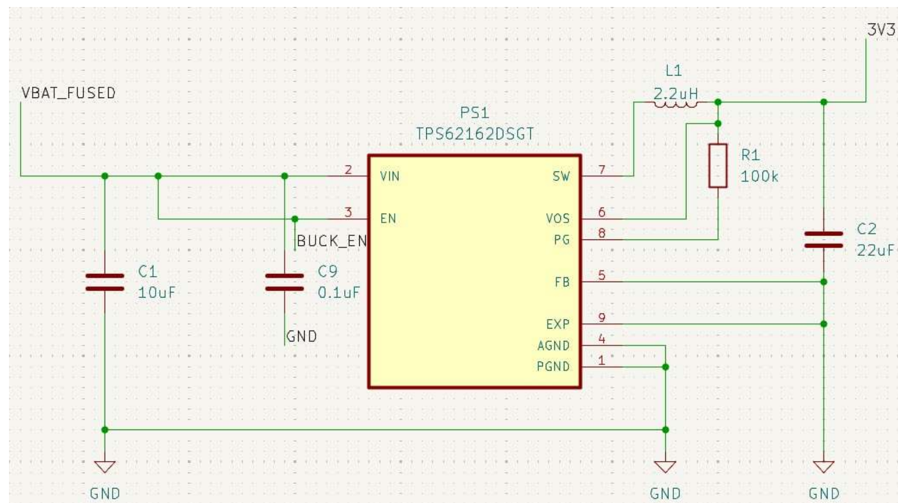

Figure 6: Buck Converter Sub-Schematic

We utilized the TPS62162DSGT synchronous step-down converter (PS1), as seen in Figure 6 above. This buck converter was chosen for its high efficiency, which minimizes heat generation in the wearable vest. The design strictly follows the manufacturer's typical application parameters for a 3.3 V output. A 2.2 µH inductor (L1) was selected to optimize the switching frequency and minimize ripple current. The input is decoupled with a 10 µF capacitor (C1), while the output is smoothed by a 22 µF capacitor (C2). An additional 0.1 µF bypass capacitor (C9) is placed on the enable pin (EN) to ensure

stable startup. This configuration guarantees a steady 3.3 V rail, fully isolating the sensitive microcontroller logic from voltage.

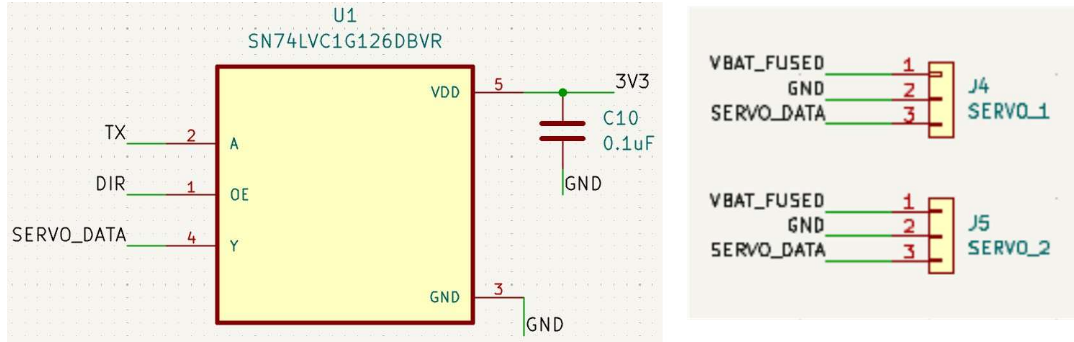

Figure 7: Bus Buffer & Servo Header Sub-Schematics

To bridge the communication, we implemented an SN74LVC1G126DBVR single bus buffer (U1) with a 3-state output, as seen in Figure 7. The ESP32's transmit line (TX) connects to the buffer's input (A), while the ESP32 controls the transmission state via the Output Enable pin (OE), labeled as DIR (Direction) in the schematic. When DIR is pulled high, the buffer drives the SERVO_DATA line; when pulled low, it enters a high-impedance state, allowing the servos to transmit data back to the ESP32. The resulting SERVO_DATA line is routed to the servo headers (J4 and J5), alongside the high-power VBAT_FUSED and GND lines. By isolating the data conversion locally on the PCB, the system can reliably drive the high-torque motors without exposing the microcontroller to logic-level mismatches.

2.4.2 Sensing

For the sensing subsystem, we investigated different sensor options for measuring posture-related body movement and determined that conductive rubber cord was the best fit for the vest. In the final design, the conductive rubber cord was placed along the middle of the user's back, where it could stretch as the user moved between upright posture and slouching. The cord behaves as a stretch-dependent variable resistor, meaning its resistance increases when it is elongated. This made it suitable for converting physical posture changes into an electrical signal that the controller could interpret

To implement this sensing method, we paired the conductive rubber cord with a fixed resistor in a voltage divider circuit. The ESP32 would then read the divider output through its ADC. This allowed changes in sensor resistance to be converted into measurable voltage changes. The ADC reading was then used to determine whether the user was in an upright or slouched posture state.
As referenced in the Appendix, the first step is converting the raw ADC count into a voltage:

$$V_{out} = \left(\frac{ADC_{raw}}{ADC_{max}}\right) \cdot V_{ref}$$

For our design, the ESP32 uses a 3.3 V reference, so V_{ref} = 3.3 V

Then, using the voltage divider relationship, the sensor resistance can be calculated as:

$$R_{sensor} = R_{fixed} \cdot \frac{(V_{out})}{(V_{ref} - V_{out})}$$

After calculating the sensor resistance, it is compared to a baseline resistance value R_0, which represents the sensor at a reference posture. The resistance change is then found using ΔR,

and normalized as:

$$\frac{\Delta R}{R_0}$$

This normalized resistance change can be used during calibration to relate the sensor output to posture state. During calibration, the user sits in an upright posture, and the corresponding sensor reading is recorded as the baseline. A slouch threshold is then set above this baseline so that when the conductive cord stretches and the ADC value rises past the threshold, the system detects a possible slouch event.

This resistance-based calculation represented our initial design approach for converting stretch sensor readings into a physical tension estimate. If the sensor were calibrated using known applied loads, the normalized resistance change could be mapped to an approximate tension value. However, during implementation and testing, we found that directly using the ADC count provided a more reliable signal for posture classification. The ADC values showed a clear difference between upright posture and slouching, and they varied over a range that was easier to threshold in software. Because our final goal was to detect posture state rather than calculate an exact strap tension, the final code used the ADC reading directly instead of converting each reading into resistance and tension.

2.4.3 Control

Control Flow:

Our control code handles Kill mode, stretch sensor interpretation and smoothing, and motor actuation and direction. In kill mode, all processes essentially stop, and the motors unspool if they are actuating or fully actuated already so that they are loose. In our code, we define a threshold for slouching which was determined by repetitive slouch testing. If the smoothed stretch sensor reading is below this threshold, the vest enters a state of good posture, but if the reading is above this threshold, it enters a slouch state, where a 30 second timer begins until motor actuation starts. When the motors fully actuate, the controller waits for 10 seconds of consistent posture, then unspools the cabling to return to a loose state again. These operations are described in the control software flow chart in the Appendix.

Smoothing Algorithm:

To have more consistent posture status updates for our vest, we had to smooth our sensor readings to prevent outliers from affecting the status, as sometimes the stretch sensor readings had sudden dips when posture was not changing. To do this, we used the following equation:

$$S_{new} = 0.15(ADC_{raw}) + 0.85(S_{prev})$$

In this equation, 0.15 is the smoothing factor. This means that each new reading contributes 15% to the updated value, while the previous smoothed value contributes 85%. This reduced sudden spikes in the sensor data and helped prevent false slouch detection. The smoothed reading was then compared to the calibrated slouch threshold. This made the sensing subsystem more reliable by ensuring that posture decisions were based on consistent trends in the sensor data rather than brief fluctuations or noise.

Servo Control Logic:

The Servo control logic determines when the vest should reel in and tighten the straps or unravel and release tension. The ESP32 controls the two STS3215 Servo motors using UART communication. These servos move in opposite directions to reel in each strap at the same time, which ensures that both sides of the vest apply tension together.

When slouching is detected for >30s, the ESP32 commands the servos to move to the calibrated "good posture" angle, meaning that the cabling is effectively "reeled in" to the angle where there is tension in the straps. In the case of our vest, which is calibrated to one user, these values are:

Servo 1 Target: +2800 **Servo 2 Target: -2800**

After actuation is done, if good posture is held consistently by the user for 10 seconds, the ESP32 then releases the tension and unravels back to the loose state. The target values for this action are as follows:

Servo 1 Target: -2800 **Servo 2 Target: +2800**

Kill Button Logic:

In addition to our Microcontroller PCB, we have a front panel PCB which is located on the front of our vest. This PCB's overall function is to assist in ESP32 programming with the EN and BOOT buttons, but its most important function is the kill button, which is a vital safety feature for this vest. On our Main Microcontroller PCB, this button is connected to GPIO 7, on the ESP32 and is configured using INPUT_PULLUP. Essentially, this means that the pin normally reads HIGH, but when the button is pressed, the pin is pulled LOW. The schematic for this PCB is shown in the Appendix.

When pressed, the kill button is momentary. Due to this, the software uses a latched kill state. Every time the button is pressed, the killMode variable is toggled between false and true. To debounce the button signal and make sure that accidental concurrent button presses are not misinterpreted by the software, we wait 50 ms. This means that to enter kill mode, you press the kill button, and the same button is pressed to exit kill mode and get back into active mode.

2.4.4 Wireless/UI

BLE Logic:

BLE support is vital to connect the control subsystem to our wireless subsystem. Overall, the ESP32 advertises itself as APCV to surrounding devices. A service UUID, command characteristic UUID, and Data Characteristic UUID are then created, whose purposes will be discussed in a later section. After we connect to the ESP32 via Bluetooth, we can notify the app of any of these changes and also receive any commands, like toggling between modes or setting target angles. We used BLE notifications (enabled

through BLE2902 descriptor) to push posture updates to the phone immediately so that the app does not have to continuously poll for data, reducing any latency. This process is further detailed in the Appendix in the BLE flow chart.

BLE Service and Characteristics:

When setting up Bluetooth communication, we have 3 main UUIDs. The service UUID represents the overall BLE service for the vest. The command characteristic UUID has the WRITE property and is used for communication from the phone to the ESP32. This helps us send commands for "Brace" and "Active" modes and move amount updates from the app. The last UUID we created is the Data Characteristic UUID, which has the READ and NOTIFY properties, and is used for ESP32 to phone communication, and this is responsible for sending any posture and sensor updates to the app.

UUID Type	UUID	Purpose
Service UUID	4fafc201-1fb5-459e-8fcc-c5c9c331914b	Main APCV BLE service
Command Characteristic	12345678-1234-1234-1234-1234567890ac	Receives app commands
Data Characteristic	12345678-1234-1234-1234-1234567890ad	Sends posture/status data

Figure 8: UUID identification table

BLE Packet Structure:

Each BLE packet includes the information that can be visually seen in the app. This information is as follows:

- **S1:** Smoothed ADC sensor value
- **Posture:** Current Posture state
- **Mode:** ACTIVE/BRACE/KILLED
- **moveAmount:** servo movement magnitude
- **slouchThreshold1:** Adjustable Posture Threshold
- **motorActuated:** Whether vest is tightened

In the app, these values are seen in the User Interface, and an example can be seen below in figure 9:

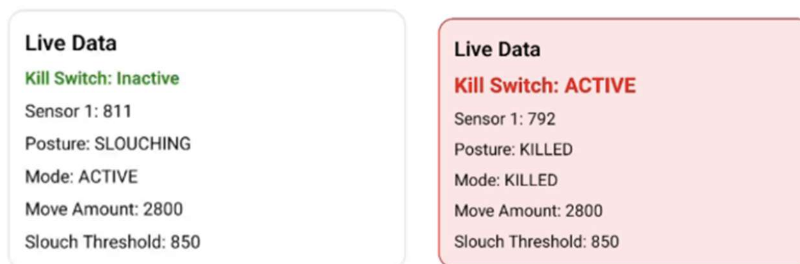

Figure 9: Portion of App Interface Displaying Live Data

App design Platform:

For our app itself, we used an Android-only interface since it facilitated app development and did not require extra cost. To develop the app, we created a custom build using React Native + Expo and the react-native-ble-px library so that our app could support BLE.

2.4.5 Physical Design

Our physical design was a huge part of this project, and we found that it was central to the success of the project. Without a solid physical design, many of our sensor readings would fail and our physical cabling system would not accurately correct the user's posture. Due to these constraints, we decided to fit and calibrate the vest to 1 person in this stage of development.

Since our initial design was thought of, we have made quite a few changes. First, we moved the front panel with our kill switch and status LEDs to the bottom of the front of our vest for easy accessibility and simpler wiring. Second, we changed the cabling material and sensor placement on the vest. We kept the same cross-back orientation for the cabling, but changed the material from elastic to ribbon, as we found that the elastic did not effectively reel in the user's shoulders, but rather just stretched itself. Along with this, we did not place the stretch sensors on the cabling itself, as the readings in this location were ineffective when trying to detect posture. Due to this, we changed the placement of this to straight down the middle of the user's back, as these readings were much more accurate. Third, we changed the placement of the PCBs on the vest, as we found that placing them on the bottom of the vest interfered the least with the mechanical placements. Lastly, we changed the spools to be machined instead of 3D printed, as we found it easier to make multiple spool iterations in testing.

These changes can be visualized in the figures below, which display the initial design versus the final implementation.

Figure 10: Front of Vest- Initial (left) versus final (right)

Figure 11: Back of Vest- Initial (left) versus final (right)

2.5 Verification

In this section, we will discuss the testing and verification of our high-level requirements. For any information on how we tested our lower-level requirements for each subsystem, refer to the Requirements and Verification table in Appendix A.

1. **Requirement 1:** The system must provide active postural restraining by applying corrective tension for 10 seconds until the user returns to a calibrated upright position, where the tension will be released after 10 seconds of consistent proper posture.

   Verification Procedure

   We first verified this by calibrating the system. This involved fitting the vest to one of our group members and getting the measurements for different posture positions. We had the user slouch as much as possible, slouch slightly, and maintain good posture, recording readings as we went by watching the serial monitor on the Arduino IDE. After getting recordings for this, we were able to set a slouch threshold which determined how our 3 states would show. To check this, we had the user slouch with the stretch sensor on the vest, and checked the Arduino IDE output for each scenario. Figure 12 below displays an example of the Serial Monitor output when the user held good posture, slouched, and when motors would actuate with a threshold of 425.

```
----------------------------
Sensor 1 Raw: 379   Voltage: 0.31
Sensor 1: Good Posture
----------------------------
Sensor 1 Raw: 537   Voltage: 0.43
Sensor 1: Slouch Detected!
Sensor 1 Slouch Time: 2 seconds
----------------------------
Sensor 1 Raw: 525   Voltage: 0.42
Sensor 1: Slouch Detected!
Sensor 1 Slouch Time: 30 seconds
Sensor 1: Slouch over 15s, motors actuating
----------------------------
```

Figure 12: Serial Monitor Output

   After initial threshold testing, we moved to adjusting the servo step count, first getting the servos moving with the FE-URT-1 controller, and then with the code itself. We set the Servo IDs to 1 and 3 in order to distinguish the two from each other and defined a MOVE_AMOUNT variable which

would determine the target angle for each servo motor. At first, we just chose an arbitrary value for this, as we wanted to see if we could make both servo motors move at the same time, in opposite directions, and to the same angle when the stretch sensors were stretched for over 30 seconds. At this point, the servos were not attached to the cabling or vest, as our goal for this portion was to see that when the user slouches pass a certain threshold, the servos actuate for 10 seconds, and after the user maintains that posture for 10 seconds, the servos release, which we verified with a timer.

Once we got basic servo movement working, we first did some initial bench testing by attaching the spool and ribbon cabling to the servo and seeing how much it took to reel in the straps completely. Once we were sure of how much the cable was reeled in by the servos, we were able to test on a user. On the app-side, we set the ability to toggle the target angle for the servos to move to. We stepped this up slowly by 500 steps using our Wireless app, starting at angle 0 to maintain a safe testing environment. After this, we found that the ideal move amount for our user was 2800 to correct the user to a “Good Posture” state. The UI for this in the app can be seen below in Figure 13.

Figure 13: App UI Displaying step toggling and Move Amount

Since we found a threshold that worked for our user, we then ran the same tests as our benchtop setup, but with the servo motors attached to the vest. We had the user slouch in the same positions, verifying the timer counts and speed of the motors needed to reach the 10 second threshold we set for ourselves.

2. **Requirement 2:** The system must be able to distinguish between poor posture and natural movement by utilizing a time delay of 30 seconds after sensing a slouch event so that it is only triggered by a prolonged period of slouching.

Verification Procedure

The slouch detection timing requirement was first verified by calibrating the system to a specific user’s upright posture and setting the slouch detection threshold based on that baseline. After calibration, the user was instructed to perform brief natural movements, such as leaning, shifting, and moving in different directions, for less than 30 seconds. During these short movements, we were able to use the Arduino IDE Serial Monitor to observe the raw sensor readings and posture state. We verified that the system correctly recognized temporary changes in posture but did not trigger corrective actuation because slouch position was not maintained for long enough. The user then held slouch positions and a timer measured how long the slouched position was sustained. As shown in Figure 14, the Arduino IDE output displayed the sensor reading, detected slouch state, and a timer displaying the duration of the slouch in seconds. The output shown in the figure uses a 15-second actuation threshold because it was captured during an earlier testing version. However, in our final implementation, this threshold was updated to 30 seconds to match the design requirement. This testing process verified that the system could filter out short natural body movements while still responding appropriately to sustained poor posture

Arduino IDE Output:

```
----------------------------
Sensor 1 Raw: 537   Voltage: 0.43
Sensor 1: Slouch Detected!
Sensor 1 Slouch Time: 2 seconds
----------------------------
```

```
----------------------------
Sensor 1 Raw: 525   Voltage: 0.42
Sensor 1: Slouch Detected!
Sensor 1 Slouch Time: 30 seconds
Sensor 1: Slouch over 15s, motors actuating
----------------------------
```

Figure 14: Arduino IDE Resistance Value Output

3. **Requirement 3:** The vest must support daily tracking of user behavior with a wireless interface and a mobile app which will be calibrated for the user and allow them to visualize their postural habits and to adjust their preferred mode (brace or active).

Verification Procedure

To verify app functionality, we first started by using a prebuilt app just to make sure that BLE was working on our microcontroller. We used the nRF Connect App on an Apple device to do initial Bluetooth testing. In the Arduino IDE, we set up our ESP32 as a BLE Server, using the BLE Server library that comes with the download of the ESP32library in the Arduino IDE. In nRF connect, the app is set up for the phone to act as the BLE Client, and we searched for the name that we advertised the ESP32 as, which was "APCV". Once we verified that we could see and connect to the APCV, we were able to select the R/W option in the app and see the chosen message come through, which was first set as a simple "Hello World", but then changed to actually show live stretch sensor status as shown below in figure 15.

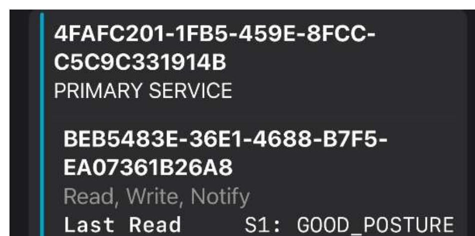

Figure 15: nRF connect data output

After verifying that the BLE Server code worked and that the actual Transceiver module on our ESP32 was sending messages, we were able to set up our own BLE environment in the app. To do this, we used a custom build of React Native + Expo and wrote our own BLE client code there. In this, we had the same Arduino IDE code, but set up a button in the app to scan for and then connect to the ESP32. When the UI for this was complete, we first clicked the "Scan for APCV" button on our app, which searched for our device. When the device was found, a new button was set to pop up, which said "Connect to APCV", which can both be seen in figure 16 below.

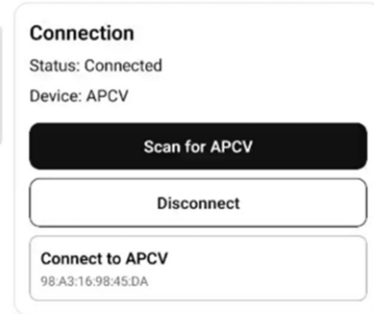

Figure 16: Scan and Connect buttons on App UI

After the scanning and connecting sequence was done, we then sent a simple packet over like what we had in nRF connect, which was a simple posture status. Once we verified that a small packet could be sent, we sent our full packet, which can be seen in figure 17 below:

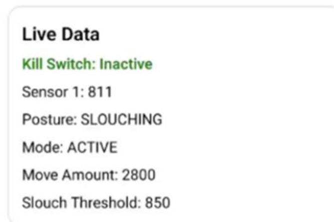

Figure 17: Data packet on UI

2.6 Costs

The total costs are displayed in Figure 18 below and they come to a total of $338.03. 5% shipping costs and 10% sales tax adds another $50.70. We will expect a salary of $45/hr * 150 hours spent * 2.5 = $16,875 per team member. If we multiply this by the number of team members, which is three, it comes to a total labor cost of $50,625

Adding our equipment cost with our labor cost, our grand total comes to $51,013.73.

Item	Type	Manufacturer	Quantity	Retail Price	Out of Pocket
ESP32-C6-DEVKITC-1-N8	Breadboard	Espressif Systems	1	$9.00	FREE
ESP32-C6-WROOM-1-N8	Main PCB	Espressif Systems	2	$11.10	$11.10
Conductive Rubber Cord Stretch Sensor	Material	Adafruit	1	$19.95	$19.95
Vest Materials	Material			$25	FREE (recycled material)
Spool Materials	Material			$10	FREE (recycled material)
Feetech STS3215 19KG 7.4V	Motor	FeeTech	2	$40.88	$81.76
Feetech STS3215 19KG 7.4V	Motor	FeeTech	1	$50.98	$50.98
2400mAh 2S 7.4V RX LiPo Battery Pack	Battery	Gens Ace	1	$27.89	$27.89
Voltage Regulator - AZ1117CD-3.3TRG1 (TO252-2)	Power & Actuation PCB	Diodes Incorporated	1	$0.27	$0.27
Capacitor - 0.1μF	PCB	Murata Electronics	5	$1.75	$1.75
Capacitor - 10μF	Power & Actuation PCB	Murata Electronics	5	$1.05	FREE
Capacitor - 1000μF	Power & Actuation PCB	Nichicon	3	$1.83	FREE
Capacitor – 0.033μF	Power & Actuation PCB	Würth Elektronik	3	$0.33	$0.33
Capacitor – 22μF	Main PCB	Murata Electronics	3	$1.38	FREE
Capacitor – 1μF	Main PCB	Murata Electronics	3	$0.45	FREE
Fuse Holder	Power & Actuation PCB	Keystone Electronics	1	$1.31	$1.31
Fuse	Power & Actuation PCB	Littelfuse Inc.	1	$0.44	$0.44
XT30PW-M connector	Power & Actuation PCB	Amass	1	$9.59	$9.59
3 Pin Connector	Power & Actuation PCB	Samtec Inc.	2	$1.50	$1.50
2.2 uH inductor	Power & Actuation PCB	TDK	1	$0.50	$0.50
3.3 V Buck Regulator	Power & Actuation PCB	Texas Instruments	2	$4.08	$4.08
Resistor - 806KΩ	Power & Actuation PCB	YAGEO	1	$0.10	$0.10
Resistor - 100KΩ	PCB	YAGEO	1	$0.17	$0.17
Resistor – 5.1KΩ	PCB	YAGEO	2	$0.20	FREE
Resistor - 0Ω	PCB	Vishay Dale	3	$0.36	FREE
Resistor - 330Ω	PCB	YAGEO	2	$0.20	FREE
Resistor - 10KΩ	PCB	YAGEO	10	$1.00	$1.00
IC BUFF 1.65V SOT-23-5	Power & Actuation PCB	Texas Instruments	1	$0.13	$0.13
IC PB ON/OFF CONTROLLER	Power & Actuation PCB	Analog Devices Inc.	1	$6.21	$6.21
2pin header	Main PCB	JST Sales America Inc.	4	$0.44	$0.44
6 pin header	Main PCB	JST Sales America Inc.	2	$0.56	$0.56
8 pin header	Main PCB	JST Sales America Inc.	2	$0.70	$0.70
USB-C	Main PCB	GCT	2	$1.60	$1.60
LED	Front Panel PCB	AEDIKO	1	$4.99	$4.99
Buttons	Front Panel PCB	Omron Electronics Inc-EMC Div	3	$1.08	$1.08
Switch - Tactile	Breadboard	Same Sky (Formerly CUI Devices)	3	$0.30	$0.30
BMS CHIP	Power & Actuation PCB	DIANN	1	$7.99	$7.99
6 position MTA100 Header	PCB	ECE supply center	3	$1.02	$1.02
22AWG Solid Yellow Hook-Up Wire 25ft	Wire	ECE supply center	1	$5.03	$5.03
PCB order		JLCDFM	3	$45.88	$45.88

Figure 18: Arduino IDE Resistance Value Output

2.7 Conclusion

The APCV successfully demonstrates an electromechanical approach to mitigating poor back posture by pairing conductive stretch sensors with servo-driven tension correction. By automatically releasing this tension once an upright stance is restored, the device actively trains users to engage their own core muscles, avoiding the atrophy often caused by traditional passive braces. Ultimately, the integration of customizable Bluetooth controls, reliable slouch-detection logic, and robust safety mechanisms provides a highly effective, user-centric solution for developing healthier long-term posture habits.

2.7.1 Ethics

We approached this project from a professional standpoint as this can affect everyday lives. Under the IEEE Code of Ethics, our first obligation is to protect and uphold public health and safety (IEEE 1.1) [1]. In terms of our posture vest, this involves acknowledging that we are placing a motorized system onto a human body. To avoid ethical breaches, we prioritized a human-centered design, as this device is meant to aid, and is not a replacement for muscular effort. It is not meant to be a device used by workers whose job requires an unrestricted range-of-motion, or by those that have serious musculoskeletal injuries that require the intervention of medical help. We have a professional duty to be honest and realistic in the claims we make, which is why we specify the audience that should and should not use this device (IEEE 1.5) [1]. Since we're putting a motorized system directly on a human body, we have a responsibility to make sure it's safe and reliable. We added a bus buffer to handle the data between the ESP32 and the motors specifically to keep the system stable. This follows the IEEE 1.5 code of ethics because it's a deliberate design choice to prevent electrical noise from the motors from messing with our sensor data or causing the software to glitch [1]. Regarding safety and regulations, the project sits at the intersection of wearable technology and assistive devices. While the FDA's General Wellness Policy provides a pathway for low-risk devices like ours to bypass the rigorous 510(k) clearance required for Class II medical devices, we align with IEC 60601-1 standards for basic safety and essential performance [2]. To mitigate the risk of mechanical over-extension, we are integrated limiters that prevent the servo motor from ever pulling beyond a safe anatomical range, regardless of what the software instructs. We also implemented a kill switch that the user can hit in case of an emergency, and it unreels the motors and shuts off the system until the button is pressed again.

2.7.2 Societal Impact

Socially and globally, the vest addresses the problems that many white-collar and office workers have with posture and "Tech Neck", potentially lowering the long-term economic burden of musculoskeletal injury and strain. By choosing materials like breathable mesh and natural fibers, we both fulfill our environmental responsibility to minimize e-waste and provide a comfortable solution for all users. Another way we made sure that the building of the vest is environmentally responsible is that we designed a modular power setup using a 2S 10A BMS and XT30 connectors so that if a part breaks, you can just swap it out instead of throwing the whole PCB and vest away. This helps cut down on e-waste. We also kept our slouch-detection algorithms open-source on GitHub so this can be a low-cost and accessible tool for any who deals with chronic posture-related pain or "tech-neck."

References

[1] IEEE, “P7-8 - IEEE Code of Ethics,” IEEE, web page. Available at: https://www.ieee.org/about/corporate/governance/p7-8. Accessed Feb. 10, 2026

[2] U.S. Food and Drug Administration, “General Wellness: Policy for Low Risk Devices - Guidance for Industry and Food and Drug Administration Staff,” FDA-2014-N-1039, Sept. 2019. [Online]. Available: https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-wellness-policy-low-risk-devices. [Accessed: Feb. 10, 2026].

[3] International Electrotechnical Commission, *Medical electrical equipment - Part 1: General requirements for basic safety and essential performance*, IEC 60601-1:2005+AMD1:2012+AMD2:2020, 2020.

[4] Division of Research Safety, “Battery Safety,” University of Illinois Urbana-Champaign, 2024. [Online]. Available: https://drs.illinois.edu/. [Accessed: Feb. 10, 2026].

[5] RobotShop, "FeeTech 7.4V 19kg Serial Bus Servo w/ Current Feedback," *RobotShop*. [Online]. Available: https://www.robotshop.com/products/feetech-74v-19kg-serial-bus-servo-w-current-feedback. [Accessed: Feb. 13, 2026].

[6] Adafruit Industries, "Conductive Rubber Cord," *Adafruit*. [Online]. Available: https://www.adafruit.com/product/519. [Accessed: Feb. 13, 2026].

[7] Shenzhen FeeTech RC Model Co., Ltd., “STS3215 Product Specification,” Ed. A/1, Jun. 23, 2023. [Online]. Available: https://files.seeedstudio.com/products/Feetech/108090023_STS3215-C001_Datasheet.pdf. [Accessed: Feb. 24, 2026].

[8] Juliette, "Measuring Stretch Forces With A Conductive Rubber Cord," *Hackster*. [Online]. Available: https://www.hackster.io/Juliette/measuring-stretch-forces-with-a-conductive-rubber-cord-d1528e. (accessed: Apr. 24, 2026).

Appendix

Full Schematics

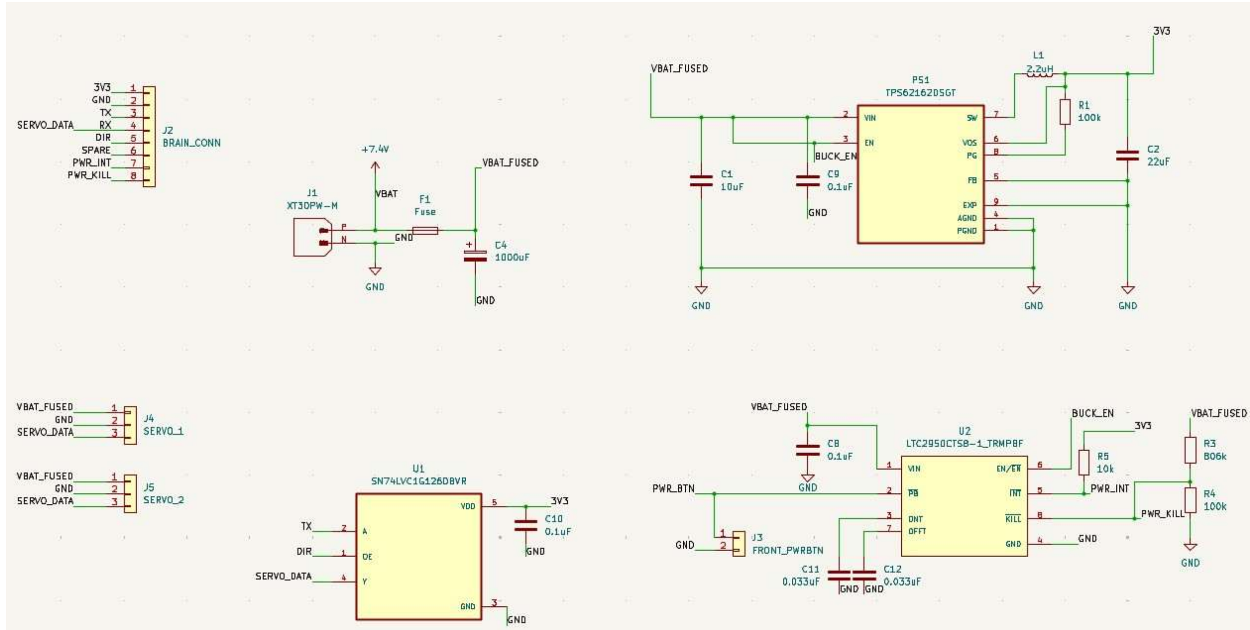

Figure A: Power & Actuation Schematic

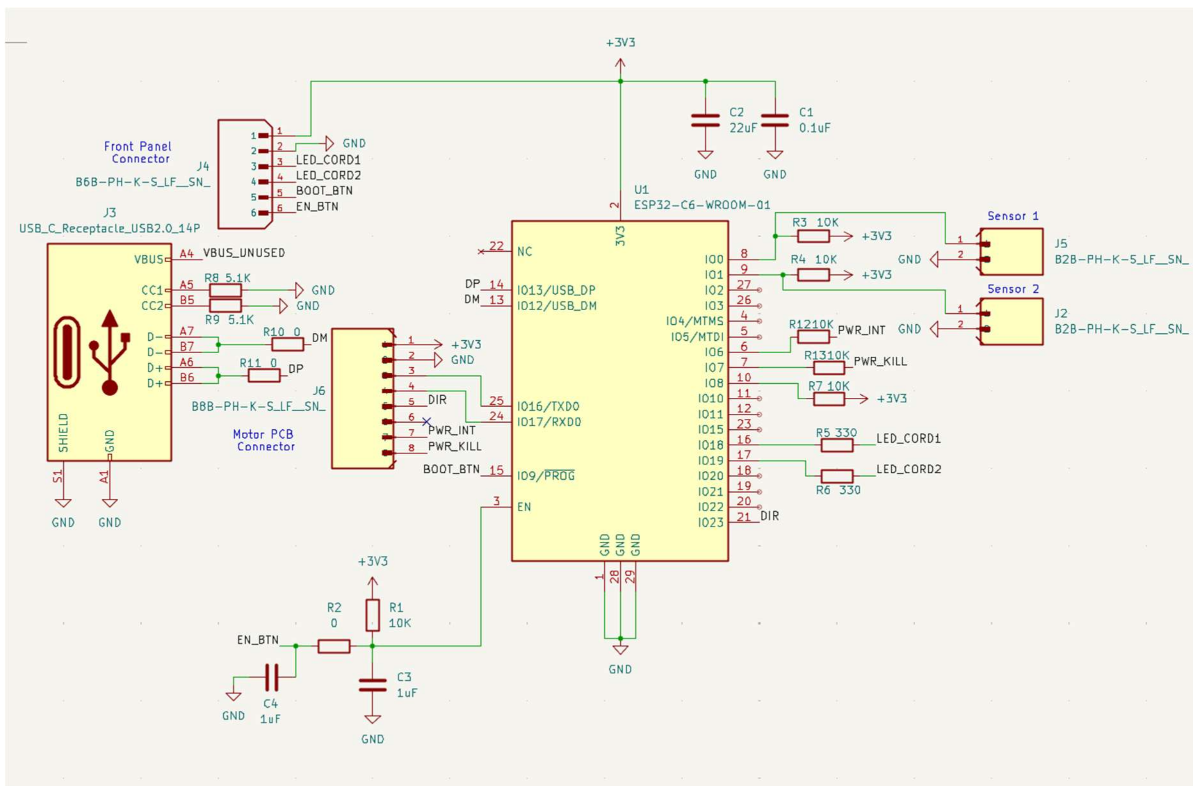

Figure B: Control Subsystem Schematic

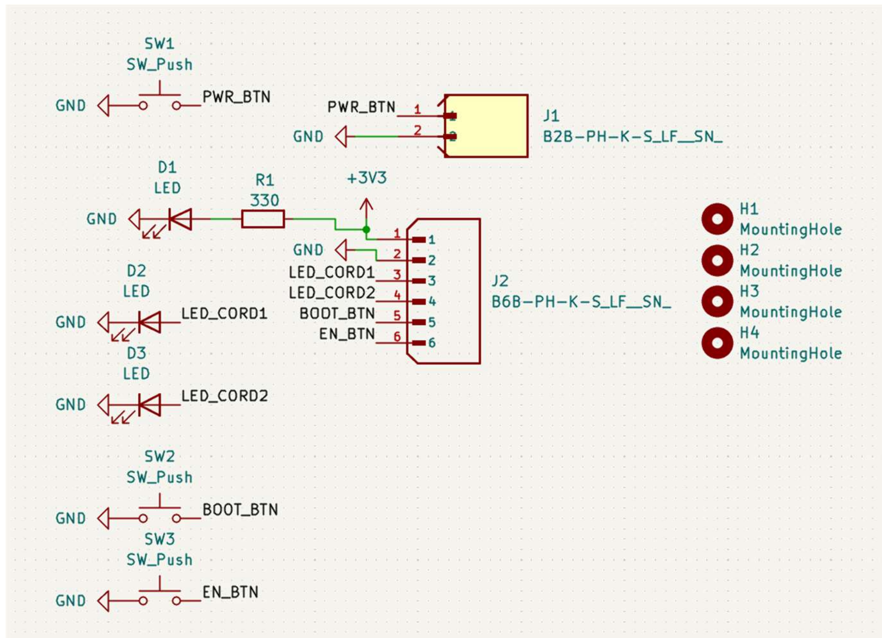

Figure C: Front Panel Schematic

PCB Board Designs

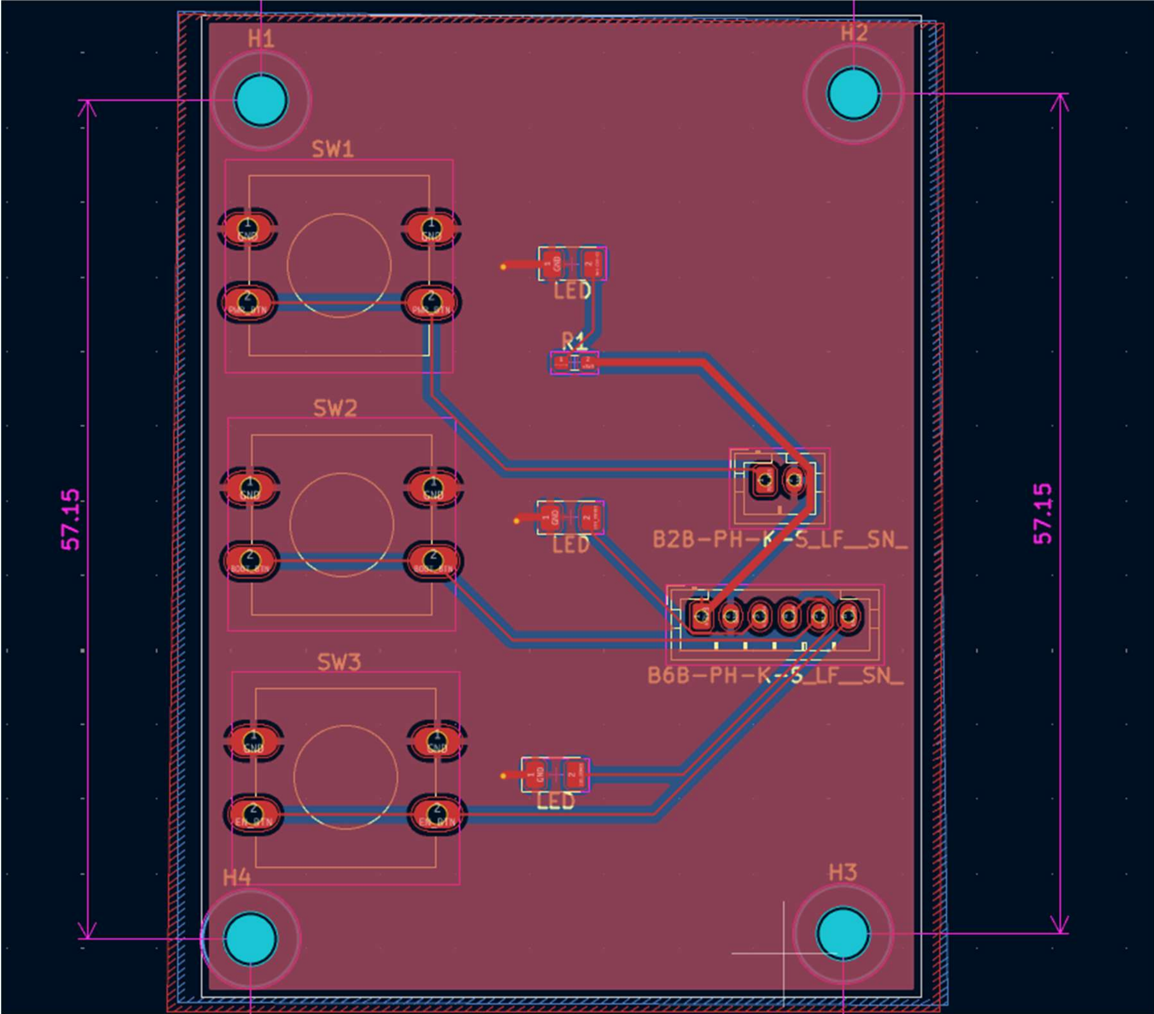

Figure D: Front Panel PCB

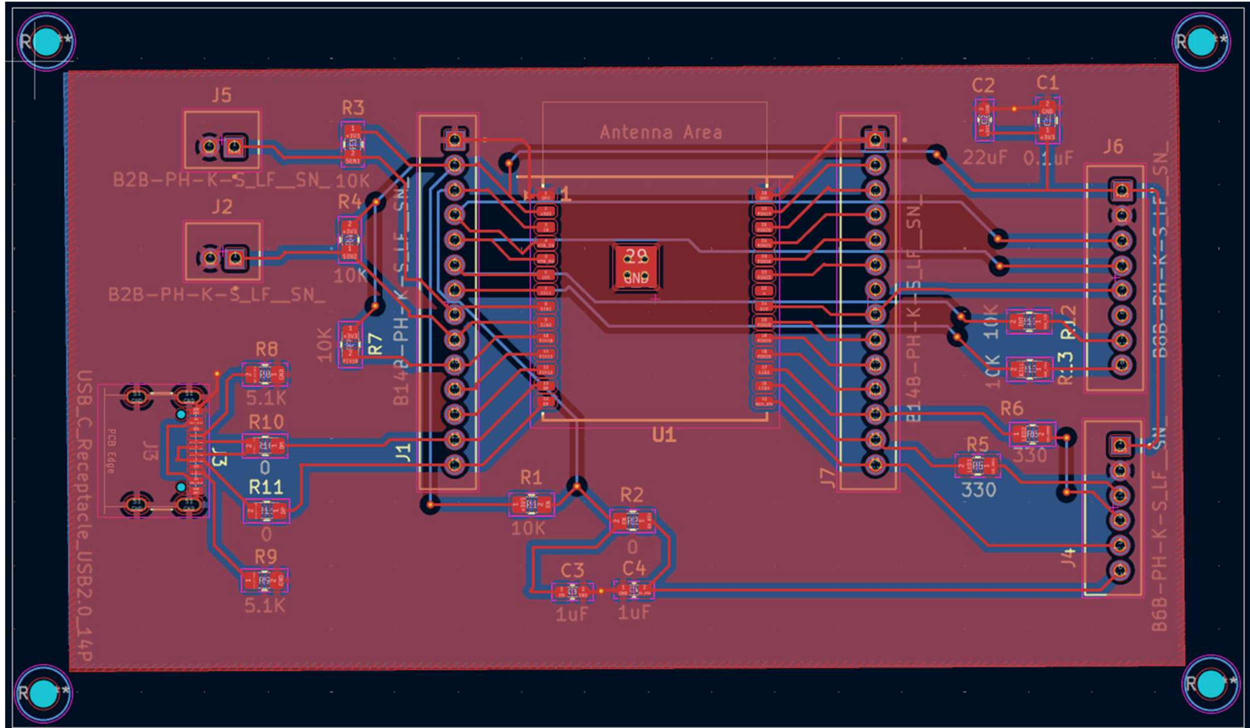

Figure E: Main PCB

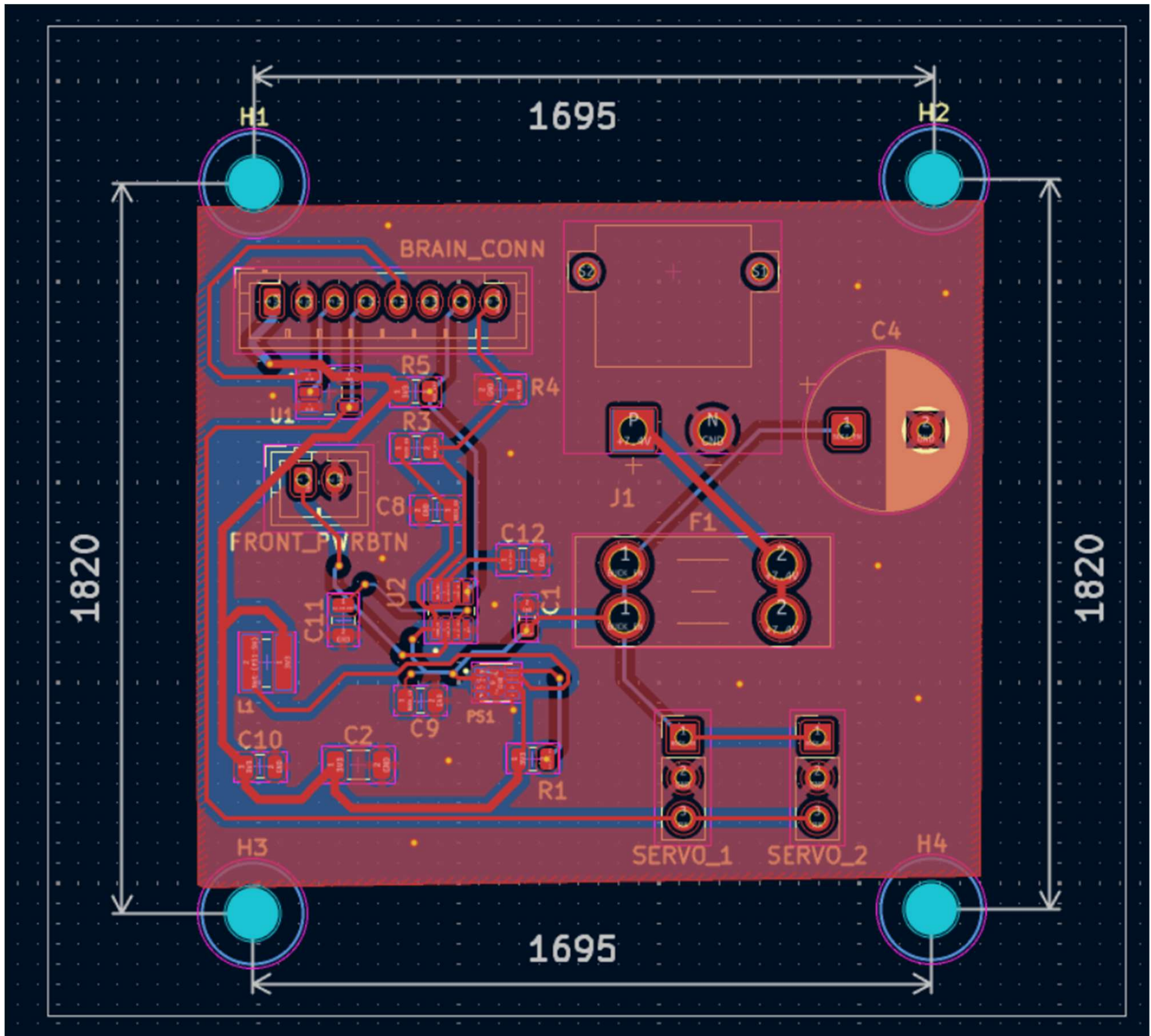

Figure F: Power & Actuation PCB

Flowcharts

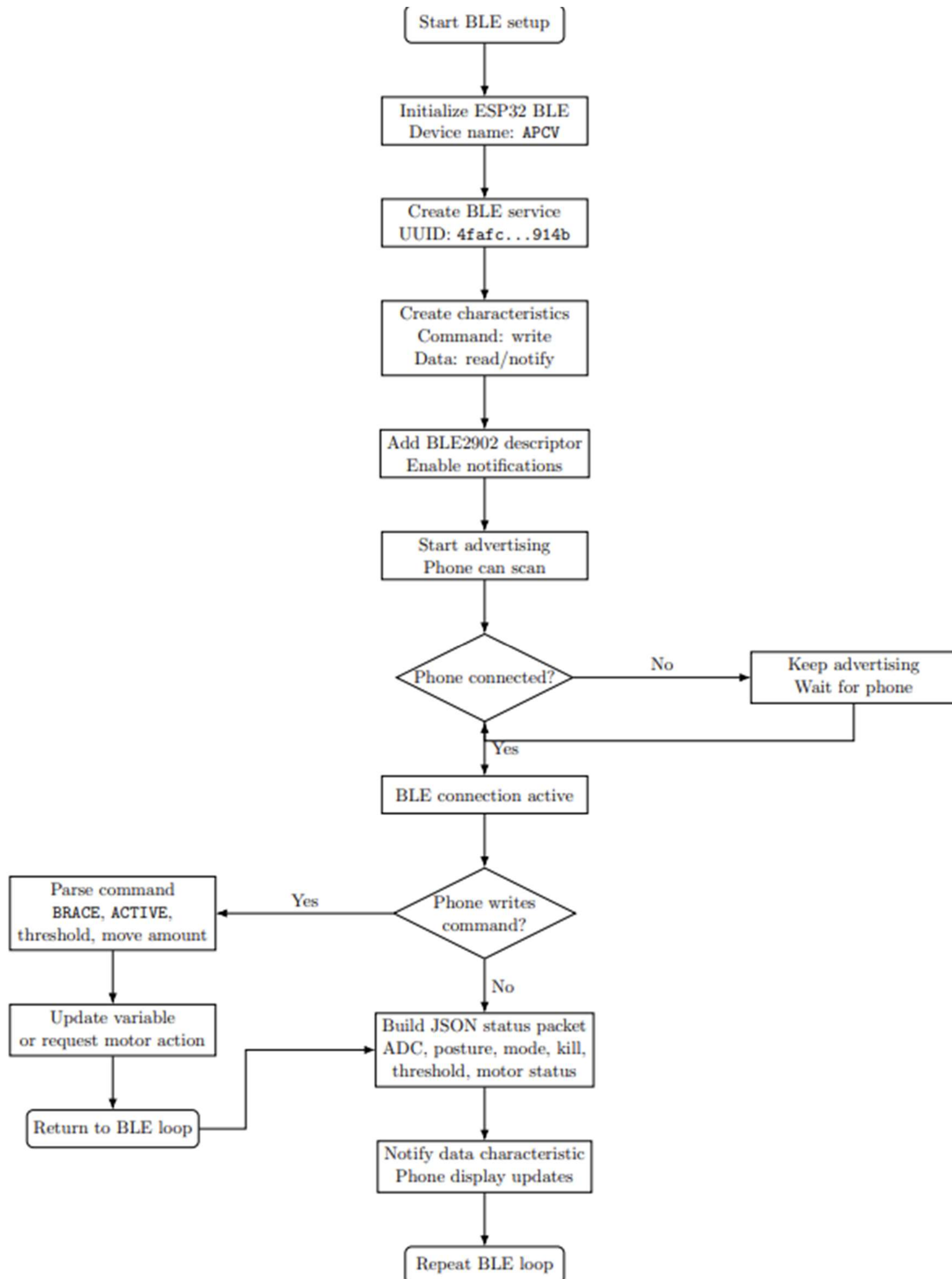

Figure G: BLE Flowchart

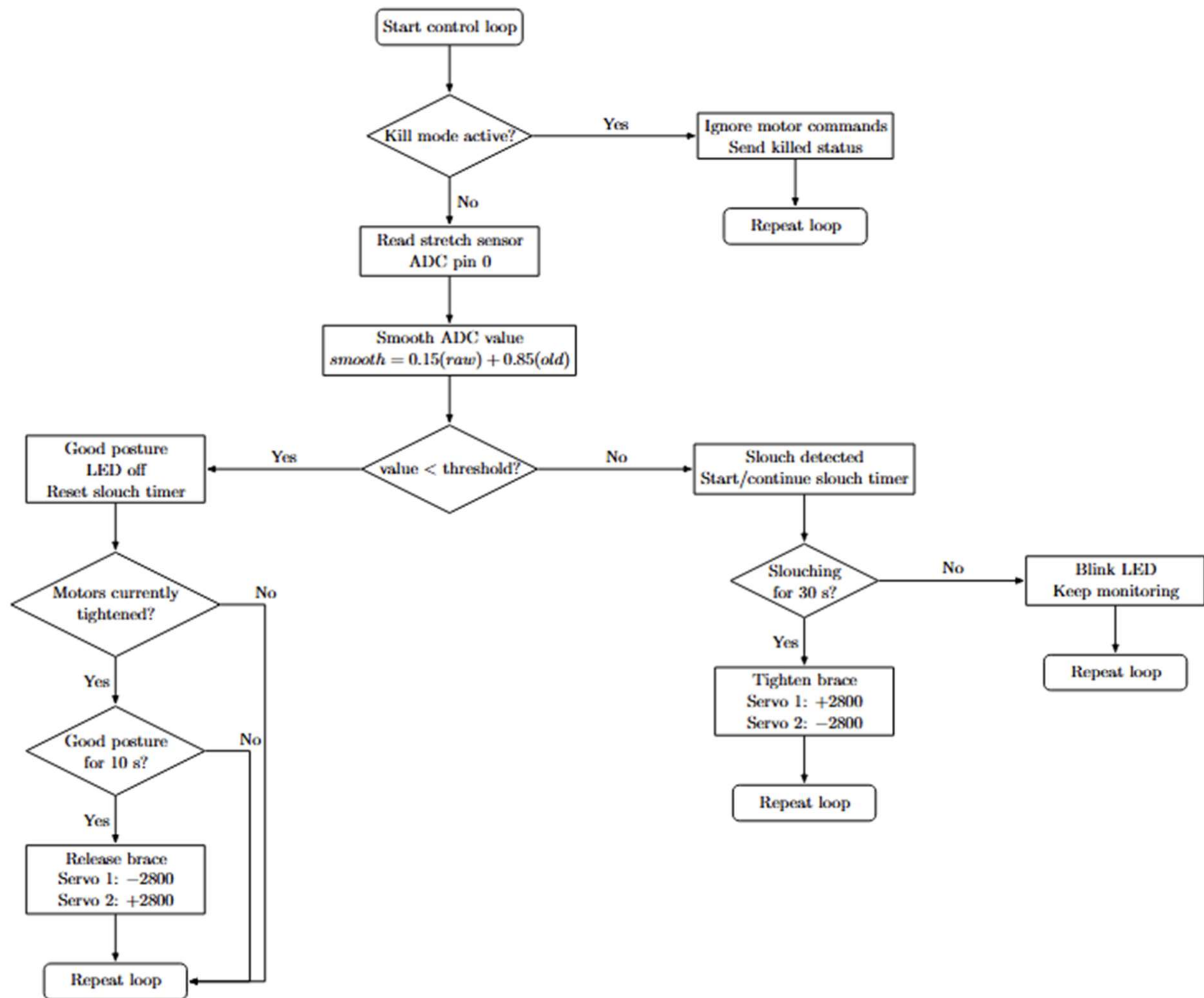

Figure H: Control Logic Flowchart

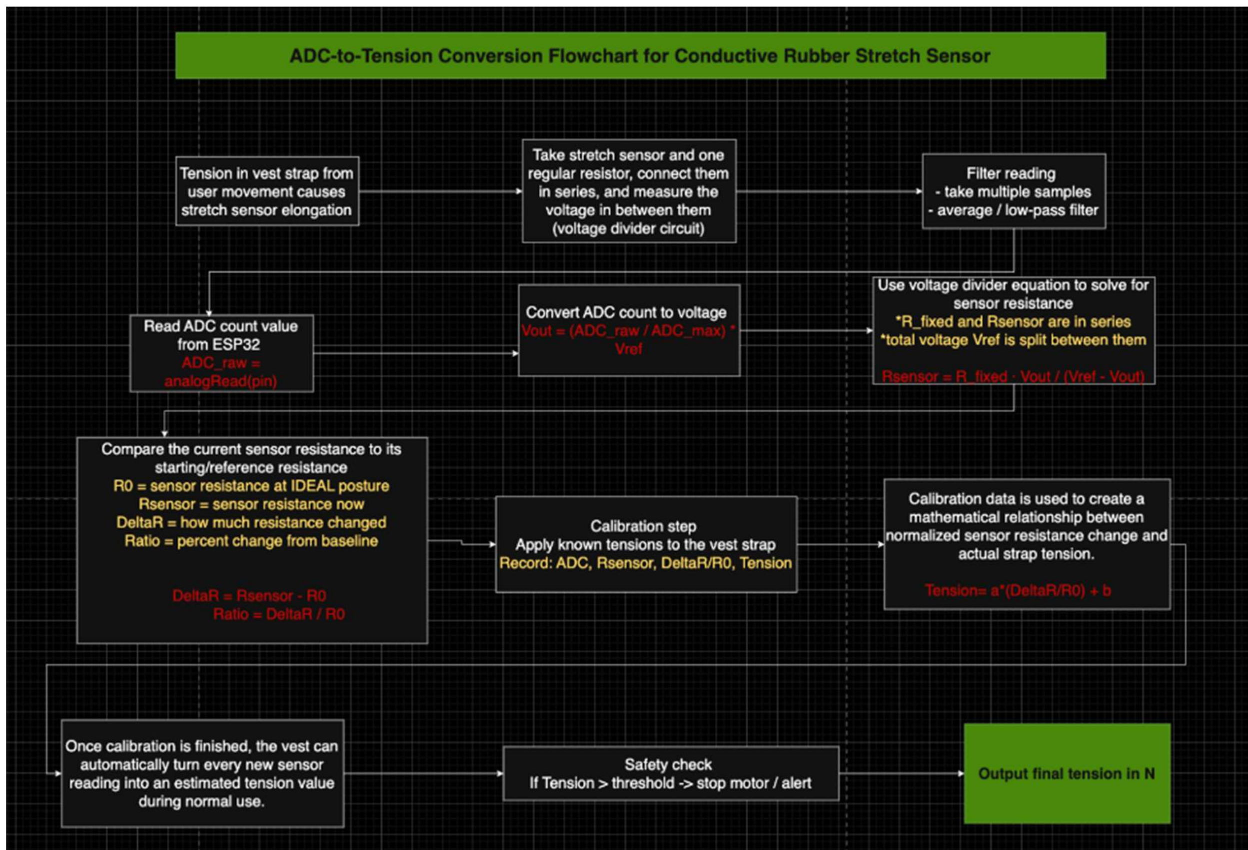

Figure I: ADC-to-Tension Conversion Flowchart for Rubber Stretch Sensor

Requirements and Verification Table

Power and Actuation Subsystem: PASS

***Note:** Tested with DC Power Supply rather than 7.4V battery due to issues with battery*

Requirements	Verification
• When the Actuation and Power Subsystem receives a signal from the Emergency Stop button on the vest, the system must immediately disconnect the 7.4V battery source from all components (Voltage Regulator, Servo Motors, MCU) within 500 ms of activation in order to fully shut the system off.	• Power on the system and make sure that ESP32 and Servos are energized. • Press the Emergency Stop. • Use a digital multimeter (DMM) to measure the voltage at the input of the voltage regulator and the power pins of the servo motors. • Confirm that the voltage that is measured is 0V within the 500ms timeframe.

• The battery pack must provide a nominal 7.4V (with range from 6.0V to 8.4V) to the Servo Motors. This will enable the motors to apply corrective tension for 10s ± 5s, then release the tension for a period of 10s ± 5s.	• We must connect the battery pack and start the posture correction software up. • We will use a stopwatch to measure the duration of the active tension after a slouch event. We must confirm that this lasts in the range of 9.5s-10.5s. • Repeat this process for the release duration. Again, it should last in the range of 9.5-10.5s. • During the active tension event, we will use a digital multimeter (DMM) to make sure that the battery voltage remains above 6V when under load.
• The Voltage Regulator must be able to step the 7.4V battery input down to a stable output of 3.3V ± 0.5V to power the ESP32 Microcontroller and stretch sensors so they do not lose power or get driven with too much voltage.	• We must connect the battery to the input of the Voltage Regulator. • We will use a digital multimeter (DMM) to measure the output voltage at the VCC pin of the ESP32 Microcontroller and at the power rail of the stretch sensors. This reading should be in the range of 3.0V-3.5V. • This must be observed for ≥60 seconds to ensure that the system is stable and that there are no brownouts or spikes in voltage.

Control Subsystem: PASS

Requirements	Verification
• The ESP32 Microcontroller must have a non-blocking 30 second timer. After detecting a slouch event, the controller must wait for the full duration of 30 seconds before applying corrective tension. The tension should only be applied if the slouching threshold persists after the delay, so if the user corrects their posture before the 30 second timer is up, the timer must reset to 0.	• We must power up the system and connect the ESP32 Microcontroller to a PC using a USB connection • We will monitor the Serial Monitor in the Arduino IDE and will set it to 115200 baud. • We will write some code to monitor what happens when we apply a load to the stretch sensor. We will add print statements to log the passing seconds after a slouch event to ensure that the timer begins and ends when it is supposed to. • We then will apply a load to the stretch sensor, which will act as our slouch

	event. We will run multiple trials, which will include: • Period of "good posture". The timer should not go off • Short period of "slouching" (<30s). The timer should go off, then stop when the slouching period stops within a threshold of 1s. • Full "slouch" event. The timer should count all the way from 0s-30s.
• The ESP32 Microcontroller must be able to continuously operate at 3.3V ± 0.1V. It must be able to continuously sample data from the Sensing Subsystem and manage the Bluetooth aspect, maintaining a code execution loop frequency of ≥ 50Hz.	• We will use a Digital Multimeter (DMM) to measure the voltage between the 3V3 and GND pins on the ESP32 during a full actuation cycle. We must confirm that the voltage is consistently 3.2V-3.4V. • To verify the loop frequency, we will use the difference between the current time and last execution time, which will be the period. We will then calculate frequency using f=1/T, and verify that this value is ≥ 50Hz

Sensing Subsystem: PASS

Requirements	Verification
• The Sensing Subsystem will use a voltage divider circuit in order to convert the variable resistance from the stretch sensors into an analog voltage ranging from 0V-3.3V. This signal will have to be linear enough to be able to distinguish good versus slouched posture.	• We will use a Digital Multimeter (DMM) to measure the voltage at the ESP32 ADC pin while the user is in an upright position with good posture. This voltage will be recorded as V_{good}. • We will then record a V_{slouch} value after having the user in a controlled slouch position (for initial testing, we will start with a more intense slouched position, then we will tune for lesser slouch positions). • We will then confirm that $\lvert V_{slouch}-V_{good}\rvert$ is ≥0.5V, as this value should confirm that the signal from the stretch sensors is strong enough to overcome noise.

• The ESP32 must be able to digitize the analog signal from the stretch cord accurately. It is vital that the raw ADC values are visible via the serial output of the Control System to ensure that the physical connection is secure enough for data to be received.	• First we will plug the ESP32 into a laptop via USB. • Now open the Arduino IDE to the serial monitor, and set the baud rate to 115200. • When observing the ADC values, we should see numbers ranging from 0-4095. • Now, we will simulate a wiring failure by disconnecting the sensor wire. When doing this, we should see a value that is either floating or ground (0 or 4095) • We will then reconnect the sensor and confirm that the values return to the 0-4095 range again.
• The conductive rubber cord must maintain a stable resistance such that the "neutral" ADC value does not drift by more than 10% over a period of 5 minutes. This will ensure that the cord is in a good condition and provides accurate data.	• The vest will be placed on the user, who must hold a good posture and stay still for 5 minutes. • We will then log the ADC values to the serial monitor on the Arduino IDE, and observe them as they come in. • After the 5 minutes are up, we will calculate the variance, which should be within a 10% range of the initial reading, or the cord should be replaced.

Wireless/UI Subsystem PASS

Requirements	Verification
• The ESP32 must maintain a stable Bluetooth connection with the smartphone at a range of up to at least 2 meters. This connection must be able to support the bidirectional transfer of posture data and calibration settings with a packet loss rate of ≤5%.	• We will first power up the vest and open up the smartphone app. • We then need to plug the ESP32 into a laptop via USB and open the Arduino IDE's Serial Monitor. • We will then move the smartphone away from the vest (or the user wearing it). • We then will use the Serial Monitor to make sure that the phone is sending calibration data to the ESP32 by putting a print statement associated with that action. After the data is sent

	from the app, the message should appear on the console.
• The UI must allow the user to easily navigate to and toggle between “Brace Mode”, where motors will maintain a pre-defined tension, ignoring timer logic; and “Active Correction Mode”, where motors will trigger after 30s of slouch detected. The ESP32 must acknowledge the changing of modes via the Serial output within 500ms.	• First, we must connect the Bluetooth app to the vest and open up the Serial Monitor on the Arduino IDE, which will be getting data from the ESP32 via USB. • To test active correction, we will select this mode on the app, check the serial monitor for a message, and slouch for <30s. The motors should not move. Then the user should slouch for >30s, and the motors should move, which will confirm that the timer logic is active. • To test brace mode, we will first select the brace mode on the app and check for a message on the serial monitor. The motors should also move to the pre-defined tension value.