

LymphSense

A wearable continuous pressure monitoring device for lymphedema

MEAM 4450/4460, MEAM Senior Design

September 2025 - April 2026

Team 02: Sonali Chandy (MEAM & IPD, schandy@seas.upenn.edu), Lindsay Fabricant (MEAM & OIDD, lfab@seas.upenn.edu), Vanessa Gong (MEAM, van3ssa@seas.upenn.edu), Lucy Liu (MEAM, lucyyyy@seas.upenn.edu), Hannah Youssef (MEAM, hyoussef@seas.upenn.edu)

Faculty Advisors: Dr. Cynthia Sung, PhD, MEAM (Associate Professor, MEAM, Soft Robotics Researcher, Sung Robotics Lab, crsung@seas.upenn.edu), Dr. Michelle Johnson, PhD (Associate Professor, PM&R, Director, Rehab Robotics Lab, johnmic@seas.upenn.edu)

Executive Summary

LymphSense is a wearable, continuous pressure monitoring device for patients with arm lymphedema. This product addresses a major need for real-time pressure feedback in the \$1 billion lymphedema treatment market. Arm lymphedema, a chronic condition that involves swelling due to build up of lymphatic fluid in the body, affects up to 40% of women treated for breast cancer post-surgery. During the first phase of intensive treatment, patients rely on physical therapists to wrap their arms using the compressive bandages. As an entirely manual and subjective process, this creates a problem that is two-fold: (1) Although physical therapists receive specialized and intensive training, they do not have a quantitative way to verify they are applying the correct pressure profile to the patient. (2) Once patients leave the clinic, they do not know if their bandaging remains in range. Moreover, if the pressure is outside of the therapeutic range, this can lead to painful swelling, resulting in prolonged treatment and an increase in clinic visits.

LymphSense delivers real-time feedback to ensure safe and personalized compression treatment. The product combines a mobile app for pressure tracking and data visualization with a sensor-integrated garment. Following a B2B hardware and SaaS model, LymphSense will be sold to hospitals, physical therapy clinics, and rehabilitation centers. Long-term, the goal is to expand into the \$4.25 billion compression therapy market and establish real-time quantitative pressure monitoring as the standard of care for compression-based treatments. LymphSense will increase efficiency, reduce costs, and provide better personalized care.

Section I: Introduction.....	5
Section I. 1: Background.....	5
Section I. 2: Problem Definition.....	5
Section I. 3: Stakeholders.....	6
Section I. 4: Existing Solutions and Limitations.....	6
Section I. 5: Social Impacts of the Solution.....	7
Section I. 6: Market Opportunity.....	7
Section I. 7: Customer Segment.....	7
Section II: Characteristics and Constraints.....	9
Section II. 1: System Characteristics.....	9
Section II. 2: Design Impact of Standards.....	10
Section III: Design, Engineering, Realization.....	11
Section III. 1 Full System Selection.....	11
Section III. 2 Sensor Subsystem.....	11
Section III. 2.1 Sensor Selection.....	11
Section III. 2.2 Sensor Design.....	12
Section III. 2.3 Sensor Manufacturing.....	13
Section III. 3 App Subsystem.....	15
Section III. 3.1 App Selection.....	15
Section III. 3.2 App Design.....	16
Section III. 3.3 App Manufacturing.....	17
Section III. 4 Final Form Subsystem.....	18
Section III. 4.1 Final Form Selection.....	18
Section III. 4.2 Final Form Design.....	21
Section III. 4.3 Final Form Manufacturing.....	22
Section III. 4 Full System Assembly.....	23
Section IV: Testing and Validation.....	24
Section IV. 1 Sensor Testing.....	24
Section IV. 2 App Testing.....	26
Section IV. 3 Final Form Testing.....	27
Section IV. 4 Full System Testing.....	30
Section V: System Performance.....	31
Section V.1: Analysis of Testing Results.....	31
Section VI: Final System Form.....	31
Section VI. 1: LymphSense Workflow.....	31
Section VI. 2: System Function and User Interaction.....	32
Section VII: Commercialization.....	31
Section VII. 1: Go-To-Market Strategy.....	31
Section VII. 2: Cost Structure.....	32
Section VII. 3: Revenue Model.....	32
Section VII. 4: IP Strategy.....	32
Section VIII: Conclusions and Future Work.....	32
Section VIII. 1: Project Effort Outcomes.....	32
Section VIII. 2: Recommendation for Future Work.....	33
Section IX: Statement of Roles.....	33
Section IX. 1: Project Components and Contributions.....	33

Section X: Acknowledgements.....	34
Figure References.....	34
References.....	35
Appendix.....	36
Section 1: Final Bill of Materials (BOM).....	36
Section 2: Revenue Model.....	37
Section 3: Code.....	37

Section I: Introduction

Section I. 1: Background

Lymphedema is a condition that involves swelling due to build up of lymphatic fluid in the body due to malfunction of lymph nodes, shown in Figure 1. Lymphedema can affect different parts of the body, including the arms, legs, face and neck. Though lymphedema can be caused by genetics, it often accompanies treatment surgeries for cancer [1].



Figure 1: Severe Arm Lymphedema in comparison with a regular arm.

The most effective management treatment for arm lymphedema is Complete Decongestive Therapy (CDT) [2]. CDT involves two phases: phase one is a four week intensive treatment period where patients regularly visit physical therapists to be wrapped using specialized techniques under compressive bandages to apply a custom pressure gradient profile to the patient's arm. The gradient pressure decreases from the hand of the patient towards the shoulder to move the fluid build up to other regions of the body where healthy lymph nodes are [3]. This bandaging keeps the swelling under control until the patient is ready for phase 2, which is the long-term maintenance phase. During phase 2, most patients are advised to wear custom-fitted compression sleeves for more than 12 hours every day [2].

Section I. 2: Problem Definition

The problem being addressed occurs during the first phase of intensive treatment. After patients leave the clinic, normal movement and daily activity can cause the pressure profile applied by the bandage wrap to change. However, there is currently no way to monitor whether the pressure remains within the therapeutic range. This creates a significant issue; if the wrap becomes too loose, it becomes ineffective, because it compromises the accuracy of custom measurements needed for the long-term management sleeve, and can ultimately worsen swelling [4].

Although physical therapists receive specialized and intensive training to administer multi-layer lymphedema bandaging (MLLB), they do not have a quantitative way to know if they are applying the correct pressure gradient profile to the patient. Existing solutions don't allow physical therapists to monitor data about the location and level of swelling in a given patient's

arm, which can be helpful in identifying a more customizable treatment plan for the patient's need.

As such, there is a critical need for a continuous pressure monitoring device for patients with arm lymphedema. LymphSense, aims to provide real-time feedback to ensure safe and personalized compression application during the intensive bandaging phase prior to compression garment measurement. The result is better personalized care while maintaining a comfortable solution.

Section I. 3: Stakeholders

The primary stakeholders for this product are the physical therapists specializing in lymphedema care and patients undergoing Complete Decongestive Therapy (CDT). Physical therapists will administer this medical device to patients.

Interviews with five physical therapists specializing in lymphedema and one lymphedema patient who has had the condition for more than 15 years have identified a need for a smart monitoring system for the pressure profile of compressive bandages. Physical therapists have communicated that they find it challenging to apply a repeatable and reliably accurate pressure gradient while wrapping patients arms, but more urgently, they've expressed how patients don't know when the wrap is no longer effective and their arms need to get rewrapped.

In addition to a smart monitoring device, the stakeholders expressed the need for an interactive method that augments the monitoring system to help them visualize pressure trends and know when specific segments of their arm have been outside of the effective pressure range. This solution would give physical therapists and patients more confidence and information during the critical intensive treatment phase.

Section I. 4: Existing Solutions and Limitations

There are currently no solutions on the market that fully solve this identified problem. The most common solution is for physical therapists to teach the patients qualitative ways to know when the pressure is incorrect. However, this method is very subjective, strenuous and doesn't provide continuous feedback to the patients.

Another existing solution is to use tension cards, shown in Figure 2. Patients can use tension cards to measure the distance between two points marked on the wrap every couple of hours to know if the pressure has changed based on how stretched the bandage is. This method is not reliable since physical therapists typically overlap multiple layers of bandage, and this method can only capture the strain in the outermost layer and not the absolute pressure. It also doesn't have the precision required for verifying the pressure gradient. Additionally, it requires a lot of effort from the patient to remember to use it, use their other hand to measure and record the measured distance each time.



Figure 2: Tension Card.

A less common existing solution is using specialized pressure monitoring sensors, like the Juzo Pressure monitoring device shown in Figure 3. While the pressure readings on this device meet the accuracy requirements, its bulky form factor makes it uncomfortable to insert in between bandage layers. Furthermore, it can only measure pressure at a single point at a given time, so it can give continuous pressure gradient readings.



Figure 3: Juzo Pressure Monitoring Device.

These existing solutions can only partially meet the needs of stakeholders, as a full solution would meet accuracy and continuous feedback needs without compromising patients' comfort. LymphSense aims to integrate a comfortable, continuous and reliable method of measuring the accurate pressure gradient with the current standard of care for arm lymphedema patients, and to improve and fit within the current workflow for the bandaging process.

Section I. 5: Social Impacts of the Solution

Lymphedema is a chronically visible condition that affects patients' confidence and self-image. Most of the existing solutions, like tension cards or pressure monitoring devices, require visible devices that probe the bandage or sit on top throughout the day. In addition to the swelling and bandage, this adds another layer of discomfort. LymphSense is personal. It is physically hidden beneath the layers of bandage and notifies the patient privately on their phone with irregular trends. It also saves them from the preoccupying thoughts of having to manually check for the pressure. They can trust that their phone will let them know if something goes wrong.

Most lymphedema patients do not have the privilege of visiting physical therapy clinics multiple times a week, and that is likely due to time, location and/ or financial constraints. LymphSense could assist those patients self-learn how to effectively and confidently rewrap themselves or with the assistance of untrained family members.

On the other side of the world, where MediCare custom-fitted compression sleeves do not exist, the bandaging treatment is not a short phase solution, but a life-long treatment method. Additionally, the American gold standard for bandage wrapping is not universal physical therapy practice. This means that patients in those countries, mainly elderly breast cancer survivors, have full ownership of their treatment. For those communities, LymphSense gives means to the patients to confidently self-learn and self-apply their bandage, for their loved ones to know how to help, and surely for untrained physical therapists around the world to self-train following released online tutorials.

Section I. 6: Market Opportunity

This project is positioned within the lymphedema treatment market and, more broadly, within the global compression therapy market, both of which are significant and growing. Lymphedema affects up to 250 million people worldwide [5]. The North America Lymphedema Treatment Market is a \$560.25 million dollar market and has a projected 11% compound annual growth rate [6]. The total addressable market is the \$4.43 billion dollar Global Compression Therapy Market [7].

Main growth drivers in these markets include increasing chronic disease rates and rising cancer survival rates. Specifically, up to 40% of breast cancer survivors develop lymphedema due to lymph node damage during surgery [8]. This contributes to an increased demand for compression-based treatment.

Section I. 7: Customer Segment

The primary customer segment this product seeks to address are physical therapists administering intensive bandaging to patients. Currently, there are 1,200 therapists certified by the Lymphedema Association of North America (LANA) [9]. These professionals follow a 135-hour intensive training to learn how to perform Complete Decongestive Therapy (CDT) for providing patient care.

This customer segment operates within hospitals, physical therapy clinics, and rehabilitation centers. Given the limited number of certified professionals and increasing demand for lymphedema care, this group represents a critical and high-impact initial customer segment.

Section II: Characteristics and constraints

Section II. 1: System Characteristics

The proposed compression monitoring sleeve must satisfy several key system characteristics to effectively support both patients and clinicians during lymphedema compression therapy. These characteristics focus on ensuring accurate pressure measurement, maintaining the therapeutic

pressure gradient along the limb, providing real-time feedback to users, and supporting comfortable long-term wear.

Criteria	System Characteristic
Pressure accuracy	Calibration accuracy within ± 5 mmHg.
Pressure gradient	Monitoring of 4 segments of the arm.
Continuous feedback	24/7 pressure data monitoring.
Comfort and usability	Breathability, no irritation and no hotspots.

Table 1: Summary of system characteristics.

Section II. 2: Design Impact of Standards

The proposed solution is intended for medical monitoring. After a conversation with an engineer at Penn Health Tech, LymphSense is classified as a Class II device. This classification has been used to guide the selection of the relevant standards as well as comparisons to similar existing solutions currently on the market.

The engineering standards and codes that apply to the solution are:

- 1. E0668:** Segmental pneumatic appliance for use with pneumatic compressor, full arm [10]. The purpose of this code is to regulate full arm lymphedema management appliances, like pneumatic compression devices. This standard has three main requirements and they are that the device should avoid the tourniquet effect, which is improper movement of the fluid, proper sizing and enclosing the entire limb.
- 2. E0672:** Segmental gradient pressure pneumatic appliance, full arm [11]. The purpose of this code is to regulate full arm sleeves that apply graduated pressure. This code mainly requires Written Order Prior to Delivery (WOPD) from physicians to the patients before use of the device.

These two standards dictate important design requirements like proper labeling, form factor, sizing, accuracy and drift for the LymphSense device.

- 3. IEC 60601-1:** Medical Electrical Equipment :Basic Safety & Essential Performance) [12]. This code requires proper insulation, handling leakage current limits, and protection against electrical shock. Additionally, it requires mechanical strength of the components, durability, thermal protection and component stability.

This standard ensures electrical and mechanical safety to avoid electric shock and ensure proper insulation of electronics.

- 4. ISO 10993-1:** Biological evaluation of medical devices [13]. This code requires toxicological risk assessment, biocompatibility tests and evaluation of exposure.

This standard ensures consideration of biological endpoints for intact-skin, prolonged contact, irritation prevention and effects of moisture.

5. **IEC 81001:** Health software and health IT systems safety, effectiveness and security [14]. Secure development, mandatory testing and cybersecurity are the main requirements for any app that handles patient data.

This standard ensures data safety, security and protection standards to keep patients' experience safe and reliable.

Section III: Design, Engineering, Realization

Section III. 1 Full System Selection

The team approached five physical therapists and after further understanding of the problem space, were consulted on proposed solution concepts.

The first concept considered was making color-changing bandages that go from green to red depending on how much strain was in the fabric. This concept would include a visual component, like LEDs or a type of embedded color-changing sensor, to indicate to the patient when the wrap is out of range. When pitched to physical therapists, it was understood that most patients are uncomfortable with the visibility of lymphedema and that having a color-changing garment as an indication would be difficult to adapt. Instead, they want a feedback method that is private and also allows them to carry on with daily activity and only pay attention to the changing pressure when needed. Additionally, they highlighted how the compression is not just dependent on the strain in the bandage but also on the overall pressure applied by other layers, like foam, and because of that a more accurate and absolute pressure reading would be embedded under those two layers rather than only at the top.

Based on that feedback, a second concept was generated, including embedding pressure sensors in the bandage layers and connecting their output to a mobile application. This concept was accepted by all of the physical therapists for its ability to maintain privacy for the patients and also alert them only when needed to take action.

After establishing the baseline solution, additional functionalities and metrics to monitor were considered, e.g. body temperature, humidity, etc. When proposed to stakeholders, they highlighted that the value of the solution is in decision-making and not just metrics monitoring. The only feature that could prompt the patient to take off the wrap or rewrap their arm is inaccurate pressure, and so as long as that one feature exists and is well done, there is no need for temperature monitoring since temperature fluctuations usually involve a lot of confounding factors, like exercise, overall body health, etc.

Section III. 2 Sensor Subsystem

Section III. 2.1 Sensor Selection

Three possible technologies for pressure sensing were taken into consideration: a tactile array sensor, a force sensitive resistor (FSR), and a pressure transducer. All of these options met the requirements light weight and low profile for form factor. Ultimately, the down selection of the sensor hardware was mainly a result of the accuracy and consistency of the sensors since each of the options met the form factor needs of the stakeholders.

Through drift and attempted calibration tests, the accuracy and consistency of the sensors were quantified. The tactile array sensor—a self manufactured item made from velostat and conductive thread, and the FSR both experienced significant drift while testing. The pressure transducer, on the other hand, demonstrated insignificant drift. The results of the pressure transducer are discussed further in section III.3.2. In addition to drift, the pressure transducer has sufficient sensitivity in the range of pressures that the stakeholders defined. The tactile array sensor and the FSR both varied significantly in output when given the same pressure input, thus making them difficult to calibrate.

Given these results, the final sensor selected for the system was a Honeywell ABP2DANT160MG2A3XX pressure transducer. This is a gauge pressure transducer with a functional operating range of 0 to 120 mmHg. The pressure transducer uses I²C protocol to transmit data and is rated to have accuracy within $\pm 0.25\%$ of the full span scale best fit line.

Section III. 2.2 Sensor Design

The design for the sensor subsystem was motivated by minimizing mass. The sensor unit consists of a pressure transducer, ethylene-vinyl acetate tubing, foam, and low density polyethylene (LDPE) pouches. The pouches are 4 cm in diameter and have a 1.5 cm diameter foam circle inside the pouch. The foam circle acts as a preset offset for the volume of the pressure sensor to make sure that there is always some initial volume in the pouches. These pouches are mounted on the sleeve and detect the pressure differential when a patient is wrapped. Attached to the pouches is plastic tubing that connects the pouch to the pressure transducer, allowing the change in pressure to be detected. The tubing is only 0.2286 cm in diameter, making it not only lightweight but low profile so that it is easily embedded into the sleeve. In total, the entire pressure sensor system is less than 1 gram in weight.



Figure 4: Assembled pressure pouch sensor.

Section III. 2.3 Sensor Manufacturing

Manufacturing each sensor subunit consists of 3 main steps: heating sealing the pouches, air sealing the tubes to the pouches, and then connecting the tubing to the pressure sensor. The pouches were made from LDPE film, foam, ethylene vinyl acetate (EVA) tubing, and glue.

In order to get a consistent and accurate manufacturing process for the first step of heat sealing the pouches, a custom made aluminum stencil was created and used. The first step begins with heating up the stamp to 300°C. A circular piece of foam is then placed between the two LDPE film layers. Once the stamp reaches 300°C, the LDPE can be heat sealed by putting a protective layer of parchment paper on top of the plastic and foam layers, then placing the stamp onto the paper and applying an even pressure on top of the stamp for 10 seconds. This results in an appropriately heat sealed pouch.

After heat sealing, the tube is then attached to the pouch through the neck that is left at the top of the circular pouch. The tube is attached to the pouch using 3M 4693H glue for plastics and left to fully cure for 24 hours. Once fully cured, the tubing is then attached to the pressure transducers with an intermediate tubing of larger inner and outer diameters to properly interface the pressure transducer with the pouch.

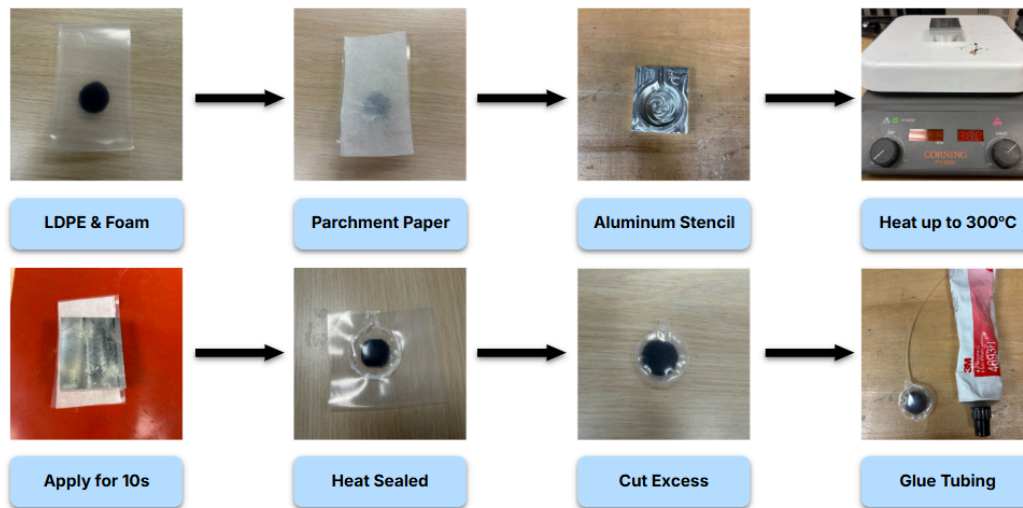


Figure 5: Sensor manufacturing workflow.

The pouches are then connected via flexible tubing to an electronics box. Inside the electronics box lies the soldered microcontroller, specifically the Adafruit QT-PY board. Additionally, the TCA9548A I2C Multiplexer was used to connect all 4 honeywell pressure transducers to the microcontroller. On two protoboards, all the components were soldered and connected to a 3.7 V Li-ion battery for power, according to the circuit diagram shown in Figure 6.

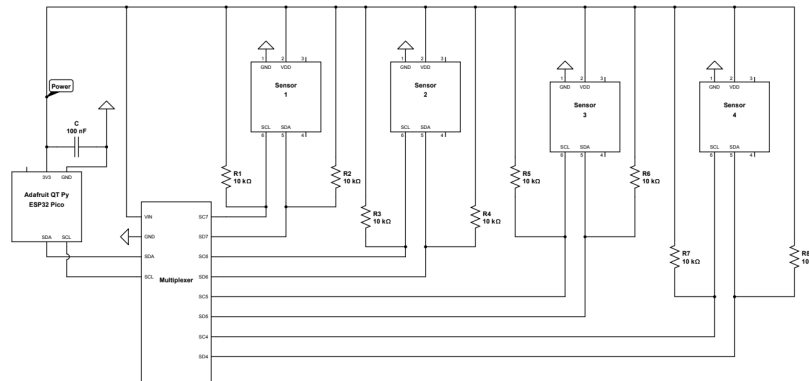


Figure 6: Circuit diagram for LymphSense electronics.

The soldered electronics, shown in Figure 7, were then placed in a 3D printed enclosure designed with a hinge mechanism for easy access.

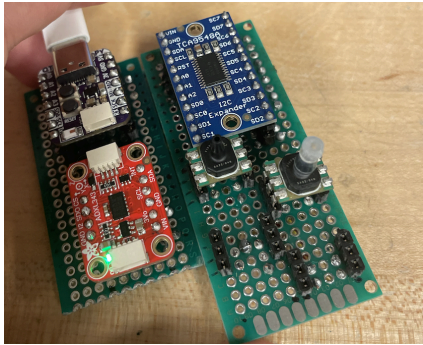


Figure 7: Protoboard layout.



Figure 8: Electronics box enclosure.

Section III. 3 App Subsystem

Section III 3.1 App Selection

To meet the stakeholder requirement of continuous feedback, color-changing short-stretch bandages, a display box, and a mobile app were considered. The first potential solution, a color-changing short-stretch bandage, was first suggested by stakeholder Colleen Maher. This solution would provide support especially for physical therapists during the wrapping process. However, this solution was not pursued as it would not meet patient needs as effectively, considering that the pressure gradient would not be visible along the bandage when worn under clothing. In addition, the second potential solution was also eliminated, a display box attached to the sensors, in which users could view pressure data. This potential solution would be bulky for patients, and a better alternative would be to make the power box as small and discreet as possible to meet stakeholder criteria of comfort. Therefore, it was determined the most effective solution to provide continuous feedback is a mobile app. This optimal solution includes color-coded in-app visuals to easily understand pressure data and reduces any unnecessary equipment to be worn by patients.

Moreover, app selection considerations included device compatibility, wireless connectivity, and general display. Stakeholders expressed preference for iOS devices over Android, so the team pursued app development in XCode. For wireless connectivity, Bluetooth maximized usage over WiFi, since short-range connections for simple device pairing was prioritized. Additionally, there were specific considerations regarding app display selection. Stakeholders expressed interest in three main display pages: a color-coded visualization page of the pressure gradient along the arm, a trends page for long-term pressure monitoring throughout the course of treatment, and a Bluetooth connection page. Lastly, push notifications support these display preferences, prompting users to open the app if pressure data is out of range or Bluetooth connection is lost.

Section III. 3.2 App Design

The app layout and general design was largely based on concepts proposed by stakeholders in order to meet user needs and design requirements. Upon opening the app, users connect to the LymphSense device via Bluetooth. This page includes a “Scan for Devices” button, which searches for nearby Bluetooth devices. If the “LymphSense Device” is found, users can connect

the device, and a circle in the top right corner of the screen on all pages will turn green and read “Connected.” Otherwise, this circle will be red and read “Disconnected.”

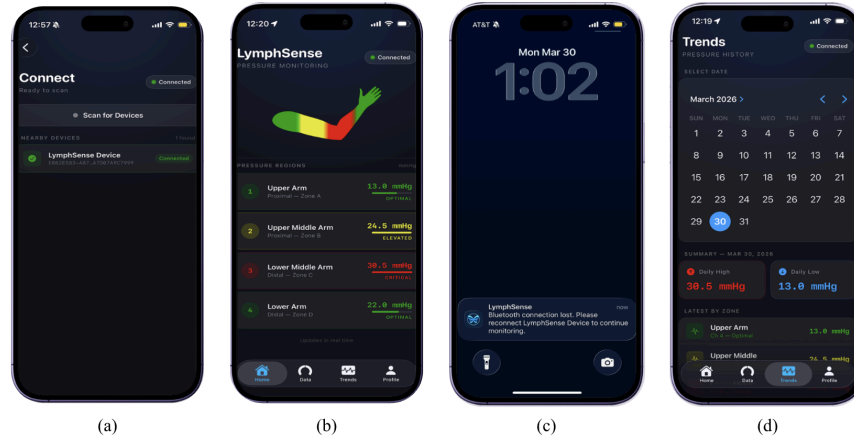


Figure 9: App Display Pages. (a) Bluetooth connection page. (b) Home page display with color-coded arm visualization of pressure gradient. (c) Push notification display. (d) Trends page display logs daily pressure data summaries.

The app Home page includes a visual of the arm with four color-coded segmented regions, where green indicates the pressure is within range, yellow if it is approaching the limit, and red if it is out of range. These pressure values mapped to specific regions of the arm are then outlined below the color-coded arm visual. If the pressure is out of range, users will receive a notification, prompting them to reopen the app and apply real-time adjustments. Push notifications appear as a banner on the iPhone with an error message.

Additionally, all pressure data is logged throughout the course of treatment to review general trends and ensure an effective pressure gradient. The Trends page includes a calendar to navigate through pressure data at different dates. For a given day, the app records daily high and daily low pressure readings, as well as latest readings by pressure segment. Users can switch through app tabs via a navigation bar at the bottom of the screen. These design choices ensure continuous, 24/7 pressure monitoring for patients and physicians.

Section III. 3.3 App Implementation

In the beginning, the app was developed based on user tutorials for Swift [15], [16]. As such, the app design is structured into user display files, Bluetooth management files, and notification management files. These files are located in Section 2 of the Appendix. User display files include *BluetoothView*, *ArmPressureView*, *ContentView*, *DetailView*, *TrendsView*, *HomeView*, *MainTabView*, and *ProfileView*. The *BluetoothView* page establishes the Bluetooth connection. *ArmPressureView* specifically generates the color-coded arm visualization. This page is supported by the *BlePeripheral*, *BluetoothManager*, and *CBUUIDs* files. *ContentView* is the structural container for all viewable pages. *DetailView* logs all data in real-time, which helps to ensure the app is properly connected. *TrendsView* displays a calendar and historical pressure data. *HomeView* includes the default page with the arm pressure visual and current pressure data.

MainTabView includes a navigation bar to change between tabs. *ProfileView* allows the user to change different settings for notifications and navigate to the Bluetooth connection page. Lastly, the *NotificationManager* file supports push notification settings. To improve overall UI and troubleshoot code errors with Bluetooth management, Claude was also used.

Section III. 4 Final Form Subsystem

Section III. 4.1 Final Form Selection

Through stakeholder interviews, skin sensitivity and comfort became critical design considerations. Patients often wear compression systems for weeks, so layers ideally need to remain aligned, breathable, and soft against the skin, while any added sensing layer should avoid creating localized pressure points.

The initial concept was to integrate the sensing system directly into a cotton stockinette layer. However, as prototyping progressed, it was realized the pressure sensors required tubing connections, and the soft structure of the stockinette could not secure the tubing without shifting, which affects sensor readings. This led to reconsideration of the feasibility of purely fabric-based form factors.



Figure 10: Cotton stockinette form factor.

After discussions with design professors at Penn, custom silicone sheets were created from scratch. Silicone initially appeared promising because of its ability to remain flexible, yet still rigid enough to embed sensors and tubings. However, during manufacturing several issues were encountered. The casting process produced inconsistent sheet thicknesses, which affected repeatability and sensor performance. The silicone layers also felt noticeably heavy on the arm, even after introducing incision patterns to reduce material usage and improve flexibility. Most importantly, the silicone remained poorly breathable, which conflicted with one of the most important stakeholder needs.



Figure 11: Silicone with incision.

Given the project involved balancing many stakeholder requirements alongside engineering constraints, weighted decision matrices were developed to guide material and enclosure selection more systematically. The first matrix evaluated candidate materials against stakeholder criteria, while the second focused on enclosure methods against engineering constraints. Both matrices are included in the appendix.

Based on the weighted decision matrix results, further exploration was conducted for the 3D-printing approach with magnetic enclosure methods, specifically experimenting with PLA-based structures with embedded fabric layers and TPU designs.

To improve breathability and comfort, both materials were printed using a customized print strategy that removed wall loops and top/bottom layers while incorporating a 10% gyroid infill pattern. This created a lightweight, porous lattice structure intended to increase breathability and flexibility.

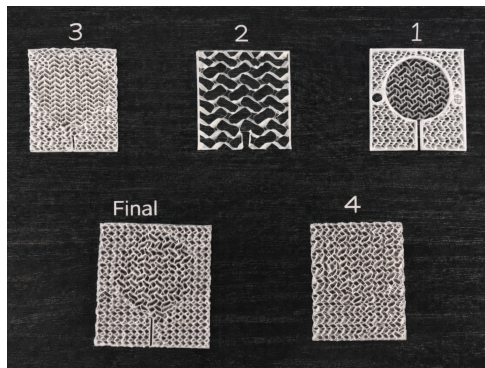


Figure 12: Print settings experimentation with TPU and PLA.

Through testing with a variety of infill densities, it was found that TPU significantly outperformed PLA. While PLA provided greater rigidity, it was too stiff to comfortably conform to the arm and accommodate natural movement. In contrast, TPU offered a softer and more compliant structure, making it more suitable for wearable applications.

However, after being professionally wrapped by an occupational therapist (OT) and evaluating the system during wear testing, the standard TPU design still did not provide sufficient comfort for long-term use. The compression layers caused visible pressure marks from both the embedded sensors and the underlying TPU lattice pattern to appear on the skin.



Figure 13: TPU full arm testing with custom print settings.

These findings ultimately motivated us to continue iterating toward softer and more breathable alternatives, leading us to explore Siraya Flex TPU Air materials with adjustable hardness ranging from 65A to 82A. By tuning different printing parameters, 65A softness was achieved, making it the softest material configuration tested so far.

Section III. 4.2 Final Form Design

Upon further down selection, the material of choice was Siraya Tech Flex TPU Air Filament. This pliable foam allowed for complete contour around the arm and is internationally recognized to be safe to use in medical devices. The LymphSense garment comprises two sheets of about 4mm in thickness, covering the forearm and the latter extending from above the elbow to the top of the bicep.

The initial construction of LymphSense also allowed for covering of the elbow. After testing led by a physical therapist at 3737 Market Street, Good Shepherd, it was advised to eliminate the elbow section of the garment. In lymphedema physical therapy, the technique to wrap the elbow is unique compared to the rest of the body, allowing for flexion and extension for “fork to mouth” movement. By eliminating the elbow section, LymphSense and relying on bandaging for movement, this goal was achieved. The placement of the four pouches included two at each piece segment, allowing for a pressure gradient to be measured between them. The physical geometry of each segment was inspired by the physical therapist textbook shapes for arm coverage, encompassing for the armpit, narrowing of the wrist, and elbow segmentation.

This sheet could be used for the entire 6-8 week wrapping period without replacement and be cut down iteratively as the patient’s arm volume decreased. Initial closure methods of each segment around the arm were explored, including magnets. These proved to not create enough force to keep the material closed and provided. After physical wear tests, it was determined that the most efficient way to close the LymphSense garment was through medical tape already accessible to the therapist alongside the general compression of the remaining bandaging layers on top.

Section III. 4.3 Final Form Manufacturing

To meet breathability expectations, the structure was 3D printed in a lattice figuration, not as a complete solid. The print includes 0 wall loops for prevention of indentations of the pouch sensors and segment boundaries. Each segment was printed at about 270 degrees Fahrenheit to achieve an optimal durometer and fit inside a 10x10inch Bambu Printer. In a mass manufacturing setting, the printer would not be a constraining factor and the physical therapist would get a larger generic rectangular piece that can be cut down to a patient's specific arm geometry.

The TPU filament for each section had a print time maximum of 90 minutes, including Z axis homing and further setup. Each print was constrained to the printing bed with each segment taking up approximately that area, as well.

Each segment needed to be split in half to accommodate the printing bed size. To glue each segment together, shoe glue and other clear-dry on adhesives were used. In a mass manufacturing setting, each segment would be one continuous piece to avoid having a stress point for potential wear. The glue adhesion did not prove to be an issue throughout testing.



Figure 14: Foam-TPU flexible garment segment for the upper arm.

Section III. 4 Full System Assembly

LymphSense is intentionally integrated to avoid any active components being under the bandage layers. For that reason, the four air-filled pouches are placed using small velcro attachments into the indentations on the flexible garment. The connecting tubes are also buried in the indentations on the garment and connected to the electronics enclosure. Based on individual preferences, the electronics box can be extended and placed in the back pocket of the user or if preferred, velcroed to the top of the arm. The four air-filled pouches are equally spaced along the garment and they are on the outer part of the arm to capture that pressure gradient in a consistent way. This attachment method for the sensors was chosen to allow for modularity. Since patients are expected to use this solution for 24/7 for 4-8 weeks and potentially again in the future, this method allows them to detach the sensors, sanitize the garment and then reconnect the pouches.

Inside the electronics box, the tubing connects the pouches to the pressure transducers which measure the differential pressure against the atmosphere and transmit that data to the mobile application via bluetooth.

Section IV: Testing and Validation

Section IV. 1 Sensor Testing

To validate the manufacturing quality of the pouch sensors as well as the short and long term accuracy of the sensors, air seal leak tests, calibration tests, and drift tests were conducted.

Three pouches that were manufactured through the process outlined earlier were randomly selected to test for seal quality. Each of the pouches were attached to a 100 mL syringe, emulating the same interfacing connections that the tubing would have with the pressure transducer. The three pouch syringe systems were kept under a pressure of 130 mmHg for 24 hours. The volume of the pouches were measured at the beginning and the end of the 24 hours by estimating the displacement of water from a graduated cylinder as shown in Figure 14.

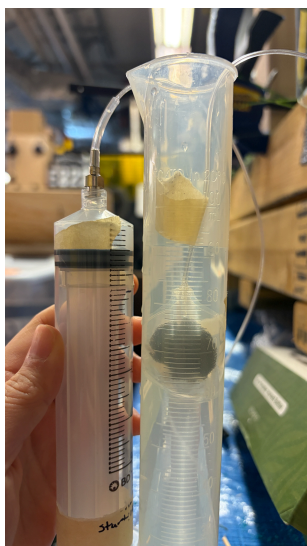


Figure 15: Air seal leak test volume measurement procedure.

This data allowed the calculation of pouch leak rate. The ASTM standard for Pressure Decay Leak Tests states an acceptable maximum leak rate of 0.0001 standard cubic centimeters per second. While each sensor resulted in slightly different leak rates, as shown in Table 4, the results still demonstrate that even in the worst case scenario, the pouches were still able to meet the ASTM F2095-07(2021) Pressure Decay Leak standard, which is a requirement for relevant devices to earn FDA approval. These tests ultimately confirmed the manufacturing quality of the pouches. Given that the start pressure range is 0-60 mmHg, the pouches survived the test with a pressure safety factor of just over 2.

	Starting Volume	Ending Volume	Leak Rate (sccs)
--	-----------------	---------------	------------------

Pouch 1	3.2 mL	3.2 mL	0
Pouch 2	4.8 mL	4.5 mL	3.4×10^{-6}
Pouch 3	3.0 mL	2.2 mL	9.26×10^{-6}

Table 4: Air leak test results

Drift tests and calibration tests from the pressure sensors validated the hardware's capability of long term accuracy and repeatability. Drift tests were conducted on all four sensors over the course of 4 hours. Each sensor was kept under a constant pressure for those four hours. After subtracting out the average signal values for each sensor, there was minimal demonstrated in all sensors as shown in Figure 16. The sensors demonstrated 0.65% drift compared to the desired detection range of 0 to 60 mmHg.

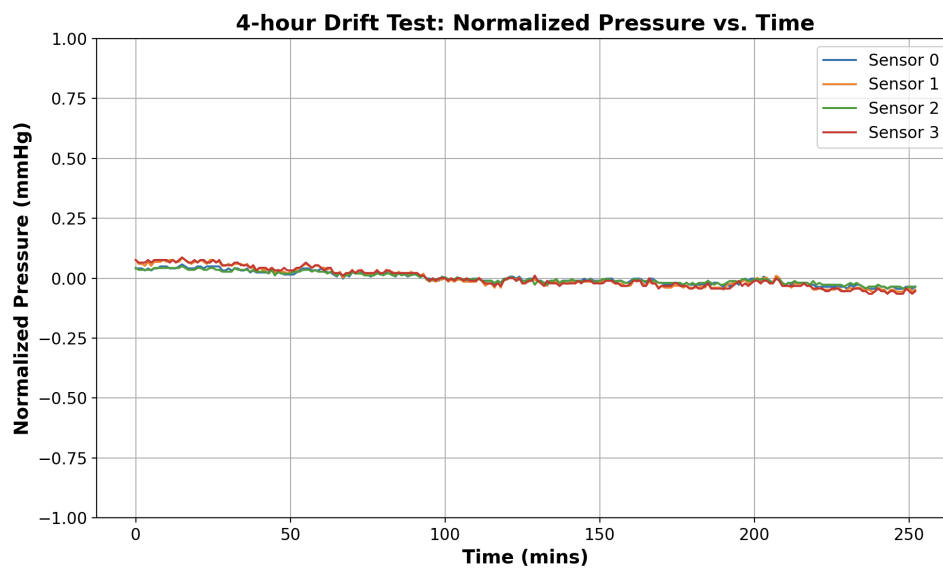


Figure 16: Sensor drift data.

Calibration of the sensors took place on a healthy volunteer's arm. The set up consisted of the cotton stockinette on the arm followed by placement of the pouch sensor and a Juzo pressure monitor sensor device next to one other in order to measure the ground truth pressure value in that specific region of the arm. Finally, a blood pressure cuff was wrapped around the arm, making sure to significantly cover the arm. Pressure measurements were taken in increments of 2 mmHg for higher resolution throughout the desired measurement range of 0-60 mmHg. Calibration equations from each 4 sensors resulted in a maximum error of 3.26 mmHg, which satisfies the stakeholder requirement of at least 5 mmHg of accuracy.

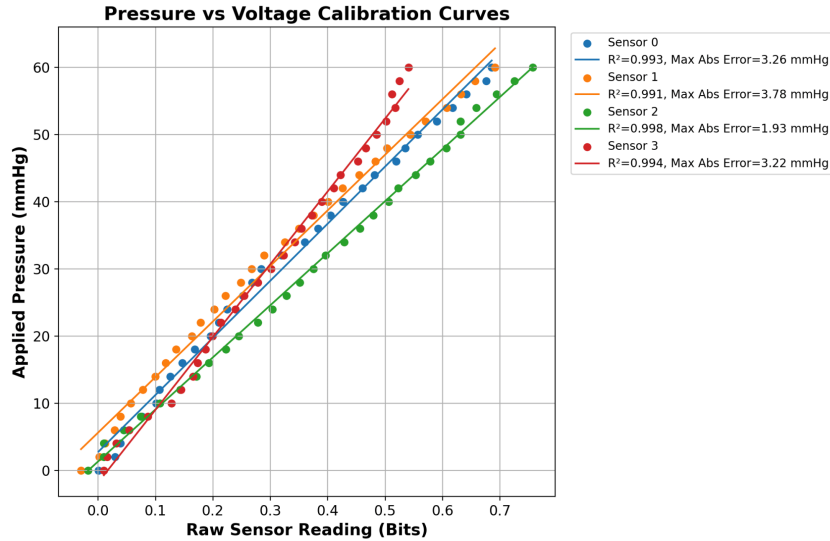


Figure 17: Sensor drift data.

Section IV. 2 App Testing

The LymphSense app has been validated for the continuous feedback stakeholder requirement. App validation tests were administered for Bluetooth connectivity, push to notifications, data management, and data visualization. To ensure effective Bluetooth connectivity, the app was tested for reliable discovery and pairing across multiple iOS devices. For a 45-minute duration test with data streamed every 1 ms, there was a 0.17% error rate. Push notifications were tested for alerts in the device foreground and background. Additionally, data was successfully saved across crash and interruption tests, which include exit app, turn off iPhone, update app, and lost Bluetooth connection. Lastly, for data visualization testing, there is increased latency at ~15,000 data points. However, since data will be saved only at 3 hour increments for continuous monitoring, the expected maximum number of data points is approximately 5,000 data points, so latency is not a concern with respect to overall app performance. As such, continuous monitoring is successfully implemented in the LymphSense app.

Section IV. 3 Final Form Testing

To evaluate whether the final garment design was suitable for long-term wear, five validation tests were conducted focusing on three key performance categories: breathability, durability, and comfort.

To evaluate breathability, a moisture vapor transmission rate (MVTR) test was conducted. In this setup, containers were filled with 70.3 grams of water and sealed using the test material samples. The sealed containers were then placed on a precision balance and monitored over 3 hours to measure mass loss caused by water vapor passing through the material. The results are recorded every 20 minutes. The three material samples are cotton stockinette, Lymphsense, and open-water. Higher mass loss corresponded to greater moisture vapor transmission and therefore better breathability.

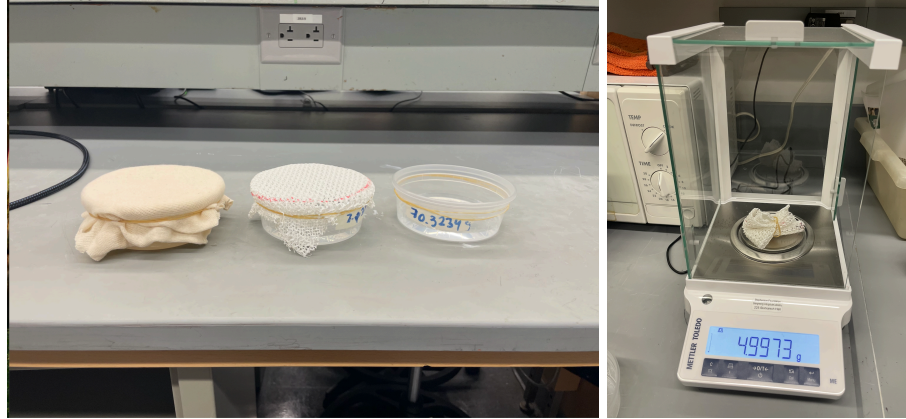


Figure 18: MVTR testing set up.

Results show that Lymphsense has a MVTR rate of 723.5 (g/m²/day) which is 1.6x higher than cotton indicating that including Lymphsense in the existing workflow will not trap additional moisture.

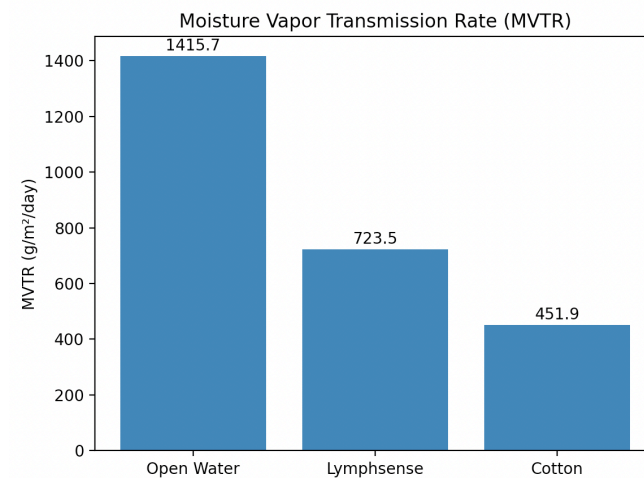


Fig 19: MVTR test results comparing 3 setup.

To evaluate durability, tensile testing was done on 10 custom-printed TPU Air dog-bone samples to characterize the material's mechanical behavior and determine the Young's modulus for each specimen. Force and displacement measurements from the MTS machine were then converted into stress-strain curves, from which Young's modulus was calculated to assess structural durability of the printed material. Results show Lymphsense has a Young's modulus of 0.11 Mpa, which is comparable to low-density foam.

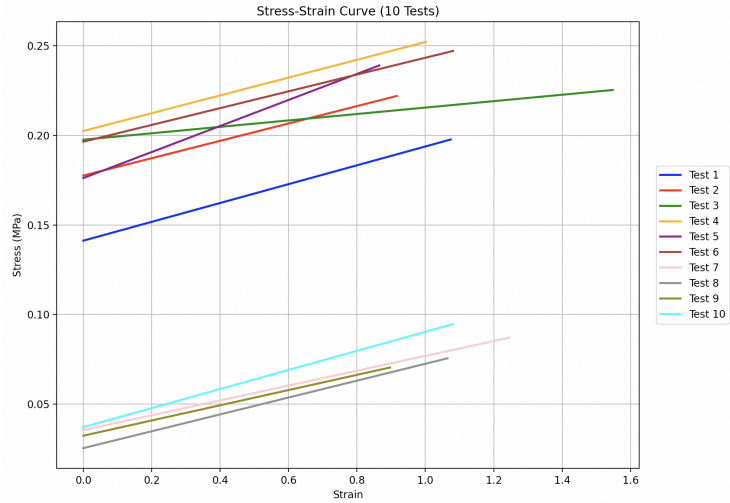


Figure 20: Stress strain curves for dog-bone samples of foam-TPU.

The purpose of the cyclic stability test was to evaluate the long-term reliability of the material under repeated loading conditions that simulate real-world wear and compression therapy use. Across five samples, each subjected to five stretch-release cycles, change in stiffness was measured with respect to time. The relatively small $\pm 10.7\%$ stiffness variation indicated minimal drift, suggesting that the material can maintain consistent performance during repeated daily use. As shown in the cyclic loading graph below, the force-displacement curves across all five cycles remain closely overlapped with minimal shifting, further indicating stable behavior over repeated use.

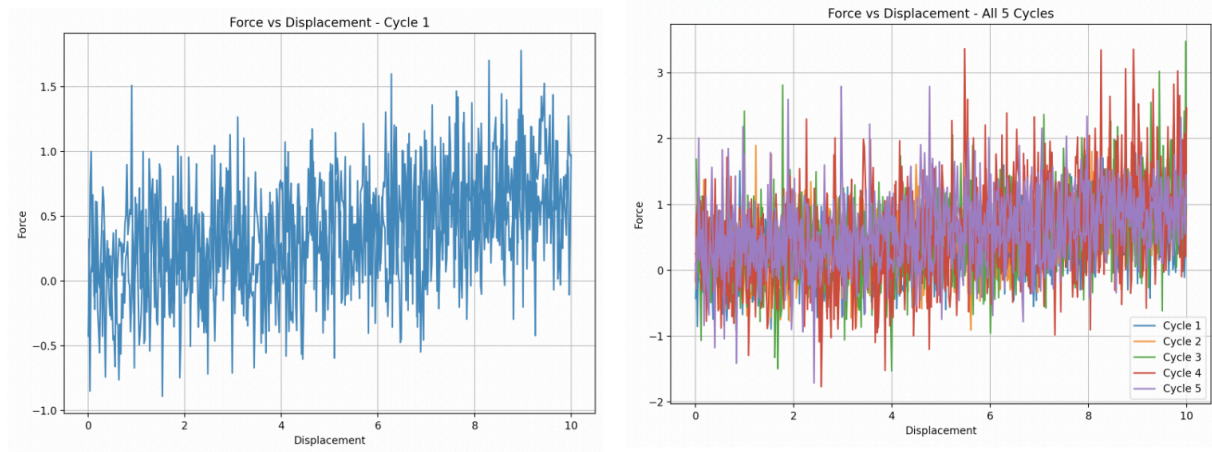


Fig 21: Cyclic loading graph for one cycle vs five cycles.

An 8-hour wearability study was conducted to assess skin comfort. Both arms were professionally wrapped by an occupational therapist, with one arm serving as the control using standard compression bandaging materials and the other incorporating the LymphSense layer.

After the wear period, the wrappings were removed and the skin was visually examined for irritation. Figure 22 shows that the arm wrapped with LymphSense exhibited minimal markings comparable to the control arm, suggesting that the sensing layer did not introduce significant additional pressure points or discomfort.

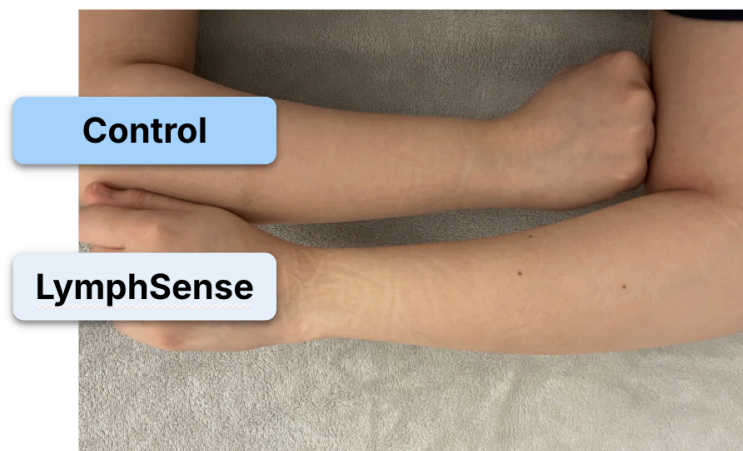


Figure 22: Residual arm markings after wrapping, without (top) and with LymphSense (bottom).

Thermal comfort was assessed using an infrared camera immediately following the wearability test. Temperature measurements were collected at the inner and outer elbow regions for both the control arm and the arm wrapped with LymphSense. Both arms demonstrated similar temperature distributions with minimal differences in average skin temperature.

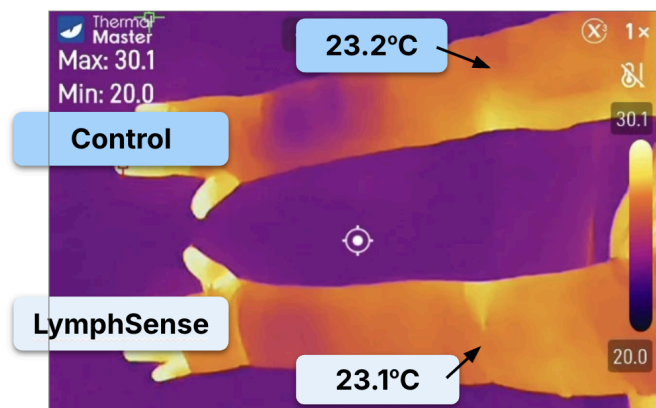


Figure 23: Thermal imaging of both LymphSense and control arm.

Section IV. 4 Full System Testing

To test the full system, Vanessa wore LymphSense around Towne and was able to connect to Bluetooth and receive notifications when it was not in the correct pressure range. To test it in the

clinical setting, Lindsay was wrapped by Alexandra Ambrosio, a Penn Medicine lymphedema physical therapist, and the entire system aided Alexandra in verifying she was applying the correct pressure gradient. These test trials were demonstrated in the full system integration [video](#). Overall, it takes 3 minutes additional minutes to include lymphsense in the standard wrapping process, which is not significant.

Section V: System Performance

Section V. 1: Analysis of Testing Results

In summary, Lymphsense successfully met all four primary stakeholder requirements: pressure accuracy, pressure gradient monitoring, continuous feedback, and comfort. Sensor calibration testing verified that the pressure transducers achieved accuracy within ± 5 mmHg, while long-term drift testing demonstrated negligible drift (0.65%) over time, confirming reliable pressure measurements. Pressure gradient monitoring was achieved through the integration of four sensing regions along the arm, allowing the app to continuously visualize pressure distribution across the limb. Continuous feedback functionality was validated through Bluetooth connectivity, notification, and data management testing, which demonstrated reliable 24/7 wireless data streaming to the mobile application. Finally, comfort requirements were validated through MVTR breathability testing, tensile and cyclic durability testing, as well as 8-hour wearability and infrared thermal imaging studies. These tests showed that the garment remained breathable, durable, and did not create significant hotspots, irritation, or additional skin markings on a healthy volunteer during extended wear.

Section VI: Final System Form

Section VI. 1: LymphSense Workflow

LymphSense is intentionally built to integrate with the current bandage treatment process. The wrapping process takes about 20 minutes total and patients may receive a lymphatic massage prior to prep the skin, also known as Manual Lymphatic Drainage or MLD. The traditional lymphedema bandaging process is layered multi functional. Starting at the skin, a cotton wrap prevents irritation, stockinette adds a soft, breathable barrier, and a foam layer ensures even pressure distribution. None of these layers apply compressive pressure.

The physical therapist will integrate LymphSense between the skin-protective cotton layers and foam and compressive bandage layers. It was initially tested to have LymphSense below the foam layer, but this was deemed unsuitable as it impaired upon the foam's main responsibility.

Although a multi-step process, the only layer being added to the bandaging process is LymphSense. All layers necessary for the current standard of care will still exist. Similar to an intake appointment, adding the LymphSense garment may add ~10 minutes to the wrapping process, though this additional time is expected to decrease until negligence with future appointments.

Section VI.2: System Function and User Interaction

The goal of LymphSense is to bring clarity to lymphedema care. The process starts by pairing LymphSense sensors to the app via Bluetooth. During a physical therapy session, the LymphSense layer is applied. As the therapist wraps, the app provides real-time feedback,

guiding the therapist to stay within the optimal pressure range. Color changes on the app show exactly where adjustments are needed. The goal is to have a fully green arm, indicating safe and effective compression.

The last layer of bandaging should also reflect a totally green arm. If sensors need recalibration, it is expected in further iterations that the patient would be able to recalibrate or seek technical help directly through the app. The electronics box is small and discreet, easily worn under clothing or clipped to a belt or pocket. In testing, the electronics box was worn in a pocket under clothing and did not create any iteration.

If a section of the garment shifts out of range, the app sends an alert. All data is logged and shared with their care team, enabling better monitoring and more predictive, personalized treatment.

Section VII: Commercialization

Section VII. 1: Go-To-Market Strategy

LymphSense will be sold directly to hospitals, physical therapy clinics, and rehabilitation centers specializing in lymphedema treatment. The go-to-market strategy involves three phases to reach national expansion. This plan leverages strong existing clinical relationships and a targeted sales approach.

Phase 1 begins in Philadelphia, home to Penn Medicine, a global leader in lymphedema care and innovation. The goal in Year 0 is to capture 10 of the 15 clinics in this region. Therapists from Good Shepherd Rehabilitation have expressed interest in serving as early adopters, which will help to build credibility and generate public interest. From there, Phase 2 will expand into the Northeast using already established University clinic relationships, including strong stakeholder relationships with therapists at Columbia University. By the end of Year 1, LymphSense projects to grow beyond Philadelphia into the suburbs, with a total 18 clinics. Based on the current model, this point represents break-even. Finally, Phase 3 will include adoption across the United States, taking advantage of regional hubs in major cities.

Section VII. 2: Cost Structure

The sensor integrated garment and device will be priced at \$500 per product. This product is priced relative to current lymphedema compression devices on the market. From the Bill of Materials, the cost is about \$235 per unit. LymphSense will seek durable medical equipment, prosthetics, prosthetic devices, orthotics, and supplies (DMEPOS) approval, allowing for the product to be sold through DME distributors. With this approval, up to 80% of the cost can be covered by Medicare for patients. Realistically, this will result in a 35-40% gross margin on hardware.

The majority of revenue, however, will come from the software cost structure. LymphSense software will be sold directly to clinics at a price of \$200 per month, which is within the standard range for remote patient monitoring (RPM) services. There is a high demand for Lymphedema bandaging treatment, as most clinics have long waitlists. Since this product has the potential to reduce overall patient treatment time by 25% or more, Lymphsense enables clinics to see more

patients, while also generating revenue through remote patient monitoring. This revenue will be driven by Remote Patient Monitoring (RPM) CPT Codes: CPT 99453, CPT 99454, CPT 99457, CPT 99458 [15].

Section VII. 3: Revenue Model

LymphSense will have a dual revenue model, as income will come from both hardware sales and software subscriptions. Annual revenue is projected to reach 9.4 million dollars within 5 years. Moreover, LymphSense anticipates break-even during Year 1, at 3,355 units, with expansion to 18 clinics.

These metrics were based on current market data. Lymphedema specialists see approximately 43 patients per month, and there are over 3,500 certified lymphedema specialists in the US [18, 19]. R&D costs are expected to be significant prior to launch and will decline over time. Hardware costs are predicted to decline as manufacturing improves, while prices are anticipated to increase with inflation. These assumptions establish LymphSense as a scalable and profitable solution.

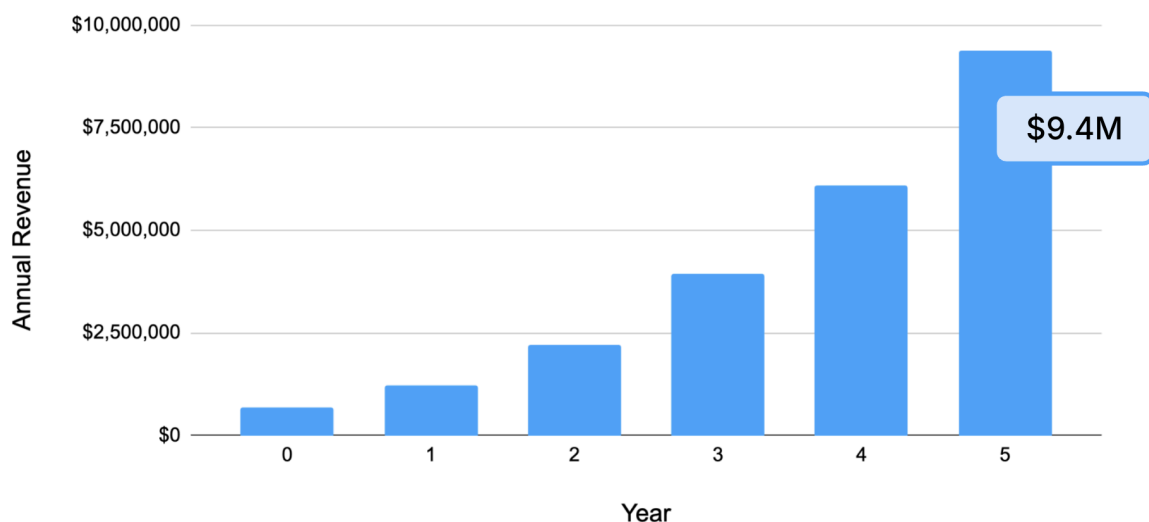


Figure 24: Projected 5-Year Annual Revenue

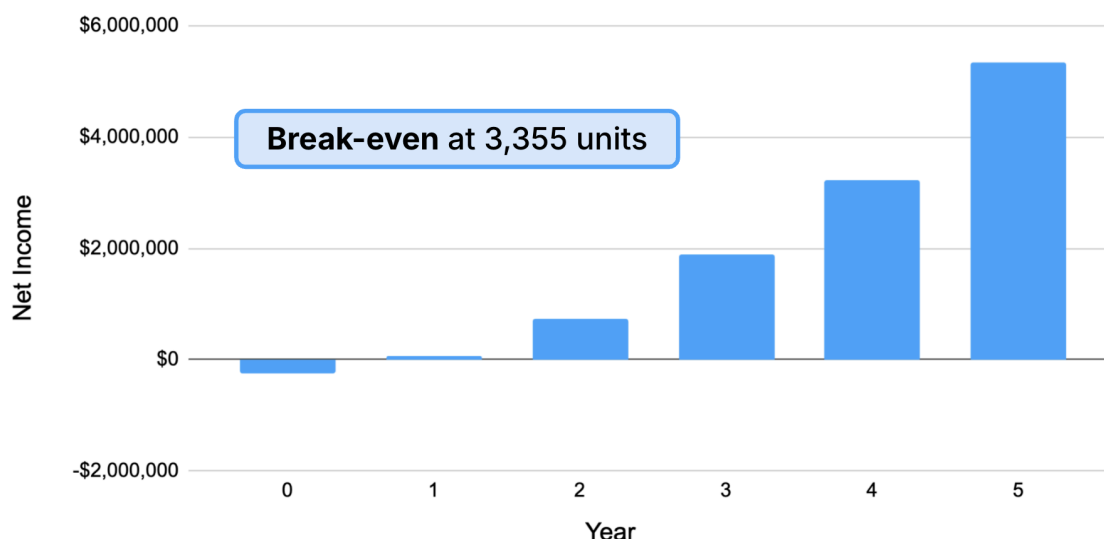


Figure 25: Projected 5-Year Net Income

Section VII. 4: IP Strategy

This technology has a strong intellectual opportunity in the novel application of a sensor-integrated 3D printed garment for pressure monitoring. Following a recent conversation with Tram Nguyen, an attorney specializing in IP and patent law, the next step will be filing a provisional patent. Additionally, the team will pursue IRB approval through Penn Medicine as advised by Dr. Ari Brooks. Dr. Michelle Johnson has advised the team to apply for grant funding through the Botswana-UPenn Partnership to support these efforts and explore the novel application of this technology in underserved communities.

Section VIII: Conclusions and Future Work

Section VIII. 1: Project Effort Outcomes

LymphSense is a proposed commercializable solution to solve a problem for an overlooked community. It addresses key needs by the customers, it has a valid business model, it starts validation testing on healthy volunteers and can be further improved towards clinical trials and full commercialization after IRB approval.

Furthermore, LymphSense can expand to sports diagnostics applications and also other types of lymphedema like lower extremity. Additionally, it can expand to applications in cardiovascular health and other areas of preventative monitoring.

LymphSense can help patients who do not have access to physical therapy self-wrap confidently. It can help patients globally who do not have access to custom-fitted compressive garments to continue the bandaging process reliably. It can help untrained physical therapists around the world self-learn how to apply bandages with real-time feedback.

Section VIII. 2: Recommendation for Future Work

The final solution was a product of 9 months of work on a new type of solution to flexible pressure monitoring, and with more time, effort and funding, there are many more outcomes and goals to accomplish. Here are a few of them:

1. IRB approval to test on patients in hospitals. Conversations with Dr. Ari Brooks, who is an Oncologist at Penn Medicine, is in progress to see the viability of this possibility. This allows LymphSense to go from R&D to an actual commercial healthcare product.
2. Iterations on hardware encasing based on user feedback to minimize size and explore more options, such as belt, for attachment.
3. Iterations on foam TPU printing given size constraints of school printers, which did not allow for printing a full garment segment and as a result segments were manually connected via heat and adhesion.
4. Testing the product on lower extremities and seeing if further iterations on form factor are needed, for example if additional anchoring is needed for the LymphSense layer.
5. Licensing the technology to med tech companies.

Section IX: Statement of Roles

Section IX. 1: Project Components and Contributions

After concept selection, the project was broken down into three big components, which are app development, sensors and electronics and flexible wearable garment engineering. Overall, Lindsay took charge of app development, Vanessa and Hannah took charge of sensors and electronics, and Sonali and Lucy took charge of wearable garment engineering. However, all team members were involved in every component of the project.

During the fall semester, collective focus as a team was figuring out the right form for the sensors and completing app development. During the spring semester, collective focus was garment engineering, sensor validation, app UI improvements and full system integration and testing. Each of these processes were extremely iterative; as there was no blueprint or an existing product to base progress off of. Productive reflection was done after each setback to think of new materials, technologies, and methodologies.

Section X: Acknowledgements

The team owes a lot of the project progress and successes to an excellent support system of technical and medical advisors: Randee Zerner, Rochelle Mendonca, Colleen Maher, Janet Kim, Jasmine Zheng, Binghao Huang, Ari Bortman, Peter Bruno, Pete Szczesniak, Jason Pastor, Sangeeta Vohra, HyJue Yoo, Ari Brooks, Clay Sulby, JeeYoon Chong, Joseph DelPreto, Audrey Mann, Ahmed Laban, Taylor Caputo, Joseph Boland, Gabriella Grym, Connor Chong, Cynthia Sung, Michelle Johnson, James Keller, Devin Carroll and Amal Youssef.

Figure References

Figure 1: Compression Store, “Compression therapy: Improve circulation, reduce swelling & support leg health.”

https://www.compressionstore.com/pages/compression-therapy?srsltid=AfmBOopxtc_wiTslpGHt2STBg-oZuOSNxX_8uFZkm4IXuVA-svfF2Na (accessed Oct. 15, 2025).

Figure 2: Medi, “circaid® Built-In-Tension® System.”

<https://www.medi.de/en/products/circaid-built-in-tension-system/> (accessed Mar. 6, 2026).

Figure 3: Design Interface Inc., “MEDICAL DEVICE GALLERY.”

<https://designinterface.com/medical-devices/> (accessed Mar. 6, 2026).

References

- [1] G. Chaput and M. Ibrahim, "Cancer-related lymphedema," *Canadian family physician Medecin de famille canadien*, vol. 69, no. 10, pp. 691–695, 2023.
<https://doi.org/10.46747/cfp.6910691>
- [2] C. Bach and A. Van Horn, "Treatment for Lymphedema: Complete Decongestive Therapy." oncolink.org.
<https://www.oncolink.org/support/side-effects/lymphedema/lymphedema-what-you-need-to-know/treatment-for-lymphedema-complete-decongestive-therapy> (accessed Apr. 29, 2026).
- [3] O. Kayıran, C. De La Cruz, K. Tane, and A. Soran, "Lymphedema: From diagnosis to treatment," *Turkish Journal of Surgery*, vol. 33, no. 2, pp. 51–57, 2017.
<https://doi.org/10.5152/turkjsurg.2017.3870>
- [4] AAPC, "HCPCS Code for Segmental pneumatic appliance for use with pneumatic compressor, full arm E0668," aapc.com.
<https://www.aapc.com/codes/hcpcs-codes/E0668?srsId=AfmBOop3No34gdkWNVf7amtwYKD52tOVHAKn4Ks-iHa4GkN9MV8NZs6I> (accessed Oct. 15, 2025).
- [5] Lymphatic Education and Research Network, "The Incidence and Prevalence of Lymphedema," lymphaticnetwork.com.
<https://lymphaticnetwork.org/living-with-lymphedema/the-incidence-of-lymphedema>
- [6] Data Bridge Market Research, "North America Lymphedema Treatment Market Size, Share, and Trends Analysis Report – Industry Overview and Forecast to 2032," databridgemarketresearch.com.
<https://www.databridgemarketresearch.com/reports/north-america-lymphedema-treatment-market?srsId=AfmBOoqXcTWKafhXuVZkXjKiYDCI81ThtI2Anco4VdyLixiltJybKmak>
- [7] Grand View Research, "Compression Therapy Market (2026-2033)," grandviewresearch.com.
<https://www.grandviewresearch.com/industry-analysis/compression-therapy-market> (accessed Apr. 29, 2026).
- [8] M. Fu, "Breast Cancer-related Lymphedema: Symptoms, Diagnosis, Risk Reduction, and Management." *World Journal of Clinical Oncology*, vol. 5, no. 3, pp. 241-7, 2014.
<https://doi.org/10.5306/wjco.v5.i3.241>
- [9] E. Anderson, "Lymphoedema Therapists: a National and International Survey," *Journal of Lymphoedema*, vol. 14, no. 1, pp. 22-24, 2019.
- [10] AAPC, "HCPCS Code for Segmental pneumatic appliance for use with pneumatic compressor, full arm E0668," aapc.com.
<https://www.aapc.com/codes/hcpcs-codes/E0668?srsId=AfmBOop3No34gdkWNVf7amtwYKD52tOVHAKn4Ks-iHa4GkN9MV8NZs6I> (accessed Oct. 15, 2025).

- [11] AAPC, “HCPCS Code for Segmental gradient pressure pneumatic appliance, full arm E0672,” aapc.com.
<https://www.aapc.com/codes/hcpcs-codes/E0672?srsltid=AfmBOoo888wmWfVd6EZJb6TNqWviKH4zA5qJtl2FApJlDiOOqNb6-fsK> (accessed Oct. 15, 2025).
- [12] International Electrotechnical Commission, “IEC 60601-1: Medical Electrical Equipment: Basic Safety & Essential Performance,” webstore.iec.ch.
<https://webstore.iec.ch/en/publication/2603>
- [13] U.S. Food and Drug Administration, “Use of International Standard ISO 10993-1, ‘Biological evaluation of medical devices - Part 1: Evaluation and testing within a risk management process,’” fda.gov.
<https://www.fda.gov/regulatory-information/search-fda-guidance-documents/use-international-standard-iso-10993-1-biological-evaluation-medical-devices-part-1-evaluation-and>
- [14] International Organization for Standardization, “IEC 81001-5-1:2021: Health software and health IT systems safety, effectiveness and security,” iso.org.
<https://www.iso.org/standard/76097.html>
- [15] Apple, “About Me,” developer.apple.com.
<https://www.google.com/url?q=https://developer.apple.com/tutorials/sample-apps/aboutme&sa=D&source=docs&ust=1777499177281542&usg=AOvVaw3VBytUb-DxuHNfLNdoZ-VT> (accessed Apr. 29, 2026).
- [16] T. Beaton and C. Cunningham, “Build A Bluetooth App Using Swift 5,” learn.adafruit.com.
<https://cdn-learn.adafruit.com/downloads/pdf/build-a-bluetooth-app-using-swift-5.pdf> (accessed Apr. 29, 2026).
- [17] American College of Physicians, “Remote Patient Monitoring Billing, Coding and Regulations Information,” acponline.org.
<https://www.acponline.org/practice-career/business-resources/telehealth-guidance-and-resources/remote-patient-monitoring-billing-coding-and-regulations-information> (accessed Apr. 29, 2026).
- [18] M. Omar et al., “Lymphedema rehabilitation: Provision and practice patterns among service providers: National survey,” *International Wound Journal*, vol. 19, no. 2, pp. 339-350, 2022.
<https://doi.org/10.1111/iwj.13634>
- [19] J. MacDonald, “An American History of Lymphoedema Management,”
<https://woundsinternational.com>.
https://woundsinternational.com/wp-content/uploads/2023/02/content_11064.pdf (accessed Apr. 29, 2026).

Appendix

Section 1: Final Bill of Materials (BOM)

General Description	Manufacturer Name	Link/Source	Subsystem
TPU sleeve	Siraya Tech Flex TPU Air - Lightweight Active Foaming Flexible 3D Filament, Adjustable Hardness 65A-82A, High-Speed Printing, Easy to Use, for Wearables, Sports Gear & Soft Prototypes (White, 1kg)	Amazon: https://www.amazon.com/Siraya-Tech-Flex-TPU-Air/dp/B0F9Y5LZTV/	Final Form
Tubes	Miniature Firm EVA Tubing for Air and Water 0.02" ID, 0.06" OD	Mcmaster Carr: https://www.mcmaster.com/1883t1/	Sensor
Sensor Interfacing Tubing	Miniature Firm EVA Tubing for Air and Water 0.05" D, 0.09" OD	Mcmaster Carr: https://www.mcmaster.com/1883T4/	Sensor
Glue	3M 5 fl oz Tube Acetone Construction Adhesive	3M: https://www.mscdirect.com/product/details/00050195	Sensor
Foam	Polyurethane Foam	University of Pennsylvania Bioengineering Labs	Sensor
Low Density Polyethylene Foam	4 Mil Poly Tubing Roll - 1 1/2" x 750'	ULINE: https://www.uline.com/Product/Detail/S-1658/Poly-Tubing/6-Mil-Heavy-Duty-Poly-Tubing-Roll-4-x-1000	Sensor
Pressure Transducer	ABP2DANT1	Digikey:	Sensor

	60MG2A3XX Honeywell I2C Pressure Transducers	https://www.digikey.com/en/products/detail/honeywell-sensing-and-productivity-solutions/ABP2DANT160MG2A3XX/21714981	
Header Pins and Prototyping Wires	–	University of Pennsylvania General Motors Lab	Electronics Box
Multiplexer	TCA9548A I2C Multiplexer:	Adafruit: https://www.adafruit.com/product/2717	Electronics Box
Microcontroller	Adafruit QT Py ESP32 Pico - WiFi Dev Board with STEMMA QT - 8MB Flash 2MB PSRAM	Adafruit: https://www.adafruit.com/product/5395	Electronics Box
3.7 V Li-ion battery	PIS-1189	Digikey: https://www.digikey.com/en/products/detail/maker-emporium-ltd/PIS-1189/13283434	Electronics Box
Protoboards	10 Pack Double Sided PCB Prototype Board, 3 x 7 cm with 240 holes	Electronix Express: https://www.elp.com/products/double-sided-breadboard-3x7cm-10pk?variant=44214613770468	Electronics Box

	Breathability	Manufacturability	Weight
Silicone	2	2.5	1
Foam	3	3	4
Neoprene	1	2	2
3d-Printed Chain Embedding Fabric In Between	3	4	3

	No overlap	Manufacturability	Elasticity	Ease of use
Zipper	4	4	1	4
Velcro	2	4	3	5
Stitch	3	2	3	1
Magnets	5	3	2	5
Buttons	1	4	2	3
Adhesive	4	4	3	4
Solid	5	2	4	2

Section 2: Revenue Model

Link:

https://docs.google.com/spreadsheets/d/1mVG8iRyXUFYsoC0ll7t_P_hZAJp9dGfnvx7DMA2ckJw/edit?usp=sharing

Section 3: Code

Section I. BlePeripheral.swift

```
//  
// BlePeripheral.swift  
// LymphSense  
//  
import Foundation  
import CoreBluetooth  
class BlePeripheral {  
    static var connectedPeripheral: CBPeripheral?  
    static var connectedService: CBService?  
    static var connectedTXChar: CBCharacteristic?  
    static var connectedRXChar: CBCharacteristic?
```

Section II. BluetoothManager.swift

```
//  
// BluetoothManager.swift  
// LymphSense  
//  
// Add this struct outside the BluetoothManager class, maybe at the top of the file.  
// MARK: - BluetoothDataPoint  
struct BluetoothDataPoint: Identifiable, Codable {  
    let id: UUID  
    let timestamp: Date  
    // 4 pressure sensor readings  
    let pressure0: Double  
    let pressure1: Double  
    let pressure2: Double  
    let pressure3: Double  
    // IMU readings  
    let imuX: Double  
    let imuY: Double  
    let imuZ: Double  
    // Device-reported time (ms)  
    let deviceTime: Double  
    private enum CodingKeys: String, CodingKey {  
        case timestamp  
        case pressure0, pressure1, pressure2, pressure3  
        case imuX, imuY, imuZ  
        case deviceTime  
        // id intentionally excluded — re-generated on decode  
    }  
    init(from decoder: Decoder) throws {  
        let c = try decoder.container(keyedBy: CodingKeys.self)  
        self.id = UUID()  
        self.timestamp = try c.decode(Date.self, forKey: .timestamp)  
        self.pressure0 = try c.decode(Double.self, forKey: .pressure0)  
        self.pressure1 = try c.decode(Double.self, forKey: .pressure1)  
        self.pressure2 = try c.decode(Double.self, forKey: .pressure2)  
        self.pressure3 = try c.decode(Double.self, forKey: .pressure3)  
        self.imuX = try c.decode(Double.self, forKey: .imuX)  
        self.imuY = try c.decode(Double.self, forKey: .imuY)  
        self.imuZ = try c.decode(Double.self, forKey: .imuZ)  
        self.deviceTime = try c.decode(Double.self, forKey: .deviceTime)
```

```

    }
    init(timestamp: Date,
        pressure0: Double, pressure1: Double,
        pressure2: Double, pressure3: Double,
        imuX: Double, imuY: Double, imuZ: Double,
        deviceTime: Double) {
        self.id = UUID()
        self.timestamp = timestamp
        self.pressure0 = pressure0
        self.pressure1 = pressure1
        self.pressure2 = pressure2
        self.pressure3 = pressure3
        self.imuX = imuX
        self.imuY = imuY
        self.imuZ = imuZ
        self.deviceTime = deviceTime
    }
}

import Foundation
import CoreBluetooth
import Combine
// --- Bluetooth Manager Implementation ---
protocol BluetoothManagerDelegate: AnyObject {
    func didUpdatePeripherals(_ peripherals: [CBPeripheral])
    func didConnect(_ peripheral: CBPeripheral)
    func didFail(error: Error?)
}
//class BluetoothManager: NSObject, ObservableObject, CBCentralManagerDelegate,
//CBPeripheralDelegate {
class BluetoothManager: NSObject, ObservableObject, CBCentralManagerDelegate,
CBPeripheralDelegate {

    private let dataFilename = "bluetooth_data_history.json"

    @Published var isScanning: Bool = false
    @Published private(set) var peripherals: [CBPeripheral] = []
    private var rssiArray = [NSNumber]()
    @Published var isConnected: Bool = false

    // @Published var receivedDataHistory: [String] = []

```

```

@Published var receivedDataHistory: [BluetoothDataPoint] = []

// Initialize the property when it is declared.
private var hourlySaveTimer: Timer?
private var lastAggregationTime: Date = Date()
weak var delegate: BluetoothManagerDelegate?
var central: CBCentralManager!
private var targetPeripheral: CBPeripheral?
private var txCharacteristic: CBCharacteristic!
private var rxCharacteristic: CBCharacteristic!

override init() {
    super.init()

    self.loadHistory()

    // Add the restoration identifier and the right options
    let options: [String: Any] = [
        CBCentralManagerOptionRestoreIdentifierKey: "LymphaSense.centralmanager",
        CBCentralManagerOptionShowPowerAlertKey: true
    ]

    // Use the options when initializing the central manager
    central = CBCentralManager(delegate: self, queue: .main, options: options)

    // Set the initial aggregation time to now (or load from disk if required)
    self.lastAggregationTime = Date()

    // Run the one-time retroactive aggregation here! fix this another time
    // self.retroactive3HourAggregation()

    // Start the hourly save timer
    startHourlySaveTimer()
}

// Invalidate the timer when the object is destroyed
deinit {
    hourlySaveTimer?.invalidate()
}

```



```
// --- Hourly Aggregation Timer ---
```

```
private func startHourlySaveTimer() {  
    // Invalidate any existing timer first  
    hourlySaveTimer?.invalidate()  
    // Schedule the timer to fire every 60 minutes (3600 seconds)  
    // For testing, change 3600.0 to 60.0 (1 minute) or less.  
    hourlySaveTimer = Timer.scheduledTimer(  
        interval: 3600.0, // 10800.0, // 3 hours // 3600.0, // 1 hour  
        target: self,  
        selector: #selector(aggregateAndSaveHourlyData),  
        userInfo: nil,  
        repeats: true  
    )  
    print("Hourly aggregation timer started.")  
}  
private func stopHourlySaveTimer() {  
    hourlySaveTimer?.invalidate()  
    hourlySaveTimer = nil  
    print("Hourly aggregation timer stopped.")  
}
```

```
// --- Persistence Methods ---
```

```
// 1. Get the URL for the save file
```

```
private func getFileURL() -> URL {  
    // Use the Documents directory for reliable app data storage  
    let paths = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)  
    return paths[0].appendingPathComponent(dataFilename)  
}
```

```
// 2. Load data from disk
```

```
func loadHistory() {  
    let fileURL = getFileURL()  
    do {  
        let data = try Data(contentsOf: fileURL)  
        let decoder = JSONDecoder()  
        self.receivedDataHistory = try decoder.decode([BluetoothDataPoint].self, from: data)  
        print("Successfully loaded \(self.receivedDataHistory.count) historical data points.")  
    } catch {
```

```

        // This is normal the first time the app runs or if the file is empty/missing
        print("No saved history found or load error: \(error.localizedDescription)")
        self.receivedDataHistory = []
    }
}

// 3. Save data to disk
func saveHistory() {
    let fileURL = getFileURL()
    do {
        let encoder = JSONEncoder()
        // Set output formatting for readability (optional)
        encoder.outputFormatting = .prettyPrinted

        let data = try encoder.encode(self.receivedDataHistory)
        try data.write(to: fileURL)
        print("Successfully saved \(self.receivedDataHistory.count) data points.")
    } catch {
        print("Error saving history: \(error.localizedDescription)")
    }
}

func clearHistory() {
    let fileURL = getFileURL()
    let fileManager = FileManager.default

    do {
        // 1. Check if the file exists before attempting to delete it
        if fileManager.fileExists(atPath: fileURL.path) {
            try fileManager.removeItem(at: fileURL)
            print("Successfully cleared (deleted) historical data file.")
        } else {
            print("No historical data file found to clear.")
        }
    }

    // 2. Clear the in-memory array and notify SwiftUI
    DispatchQueue.main.async {
        self.receivedDataHistory = []
        print("In-memory history array cleared.")
    }
}

```

```

    } catch {
        print("Error clearing historical data file: \"(error.localizedDescription)")
    }
}
// new
// MARK: - Pressure Alert Helper
private let pressureTargets: [(low: Double, high: Double, tolerance: Double, name: String)] =
[
    (18, 22, 1, "Upper Arm"),
    (23, 27, 1, "Upper Middle"),
    (28, 32, 1, "Lower Middle"),
    (33, 37, 1, "Lower Arm"),
]
private var lastPressureAlertTime: Date = .distantPast
private func checkPressureAlerts(for point: BluetoothDataPoint) {
    let now = Date()
    guard now.timeIntervalSince(lastPressureAlertTime) >= 120 else { return }
    let readings = [point.pressure0, point.pressure1, point.pressure2, point.pressure3]
    var outOfRange: [String] = []
    for (i, value) in readings.enumerated() {
        let spec = pressureTargets[i]
        let isGreen = value >= spec.low && value <= spec.high
        let isYellow = !isGreen && (value >= spec.low - spec.tolerance && value <= spec.high
+ spec.tolerance)
        if !isGreen && !isYellow {
            outOfRange.append(spec.name)
        }
    }
    if !outOfRange.isEmpty {
        let zones = outOfRange.joined(separator: ", ")
        NotificationManager.shared.schedulePressureAlert(zones: zones)
        lastPressureAlertTime = now
    }
}
// --- MARK: - State

// The centralManagerDidUpdateState is the *only* correct place to begin the scan.
func centralManagerDidUpdateState(_ central: CBCentralManager) {
    switch central.state {

```

```

case .poweredOn:
    print("Bluetooth is powered on. Starting scan now...")
    // FIX: The scan is triggered ONLY when the state is ready.
    startScan()
case .unknown:
    print("Bluetooth state is unknown (0). Waiting for update...")
case .poweredOff:
    print("Bluetooth is powered off (4). Please turn on Bluetooth.")
    isScanning = false
case .resetting:
    print("Bluetooth connection is temporarily lost (1).")
    isScanning = false
case .unsupported:
    print("Bluetooth Low Energy is not supported (3).")
    isScanning = false
case .unauthorized:
    print("The app is not authorized to use Bluetooth (2).")
    isScanning = false
@unknown default:
    print("A new state was added that is not yet handled.")
    isScanning = false
}
}
// --- MARK: - Scan (Targeted Service)

// Scan function with an added state check for robustness.
func startScan() {
    guard central.state == .poweredOn else {
        print("Cannot start scan, Bluetooth is not powered on. Current state:
\\(central.state.rawValue)")
        return
    }
    peripherals.removeAll()
    isScanning = true
    print("Scanning for Bluefruit UART...")
    central.scanForPeripherals(withServices: [CBUUIDs.BLEService_UUID])
    DispatchQueue.main.asyncAfter(deadline: .now() + 10) { [weak self] in
        self?.stopScan()
    }
}
}

```

```

// --- MARK: - Scan (General BLE)

// General scan function with an added state check for robustness.
func scanForBLEDevices() {
    guard central.state == .poweredOn else {
        print("Cannot start general scan, Bluetooth is not powered on. Current state:
\\(central.state.rawValue)")
        return
    }

    peripherals.removeAll()
    rssiArray.removeAll()

    central.scanForPeripherals(withServices: [] , options:
[CBCentralManagerScanOptionAllowDuplicatesKey:true])
    print("Scanning for all BLE Devices...")
    DispatchQueue.main.asyncAfter(deadline: .now() + 10) { [weak self] in
        self?.stopScan()
    }
}
func stopScan() {
    central?.stopScan()
    isScanning = false
    print("Stopped scanning.")
    delegate?.didUpdatePeripherals(peripherals)
}
// --- MARK: - Discovery

func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
    print("Function: \\(#function),Line: \\(#line)")
    if !peripherals.contains(where: { $0.identifier == peripheral.identifier }) {
        peripherals.append(peripheral)
        rssiArray.append(RSSI)
        delegate?.didUpdatePeripherals(peripherals)
    }

    print("Peripheral Discovered: \\("LymphSense Device" ?? "Unknown")")
}

```

```

// Convenience to match SwiftUI view call site
func connect(_ peripheral: CBPeripheral) {
    connect(to: peripheral)
}
// --- MARK: - Connect

func connect(to peripheral: CBPeripheral) {
    targetPeripheral = peripheral
    targetPeripheral?.delegate = self
    central.connect(peripheral, options: nil)
    stopScan()
}
func centralManager(_ central: CBCentralManager,
                    didConnect peripheral: CBPeripheral) {
    print("Connected:", "LymphSense Device" ?? "Unknown") // replaced peripheral.name
    delegate?.didConnect(peripheral)
    isConnected = true
    peripheral.discoverServices([CBUUIDs.BLEService_UUID])
}

func centralManager(_ central: CBCentralManager, didFailToConnect peripheral:
CBPeripheral, error: Error?) {
    print("Failed to connect to \( "LymphSense Device" ?? "Unknown")":
    \(error?.localizedDescription ?? "Unknown error")")
    delegate?.didFail(error: error)
    targetPeripheral = nil
}
func centralManager(_ central: CBCentralManager, didDisconnectPeripheral peripheral:
CBPeripheral, error: Error?) {
    print("Disconnected from \( "LymphSense Device" ?? "Unknown")":
    \(error?.localizedDescription ?? "Unknown error")")
    isConnected = false

    // Action: Send the disconnection notification
    NotificationManager.shared.scheduleDisconnectionNotification(deviceName: "LymphSense
Device")

    targetPeripheral = nil
    self.isConnected = false
}

```

// --- MARK: - Services → Characteristics

```
func peripheral(_ peripheral: CBPeripheral,
               didDiscoverServices error: Error?) {
    if let error = error {
        delegate?.didFail(error: error)
        return
    }
    peripheral.services?.forEach {
        peripheral.discoverCharacteristics(nil, for: $0)
    }
}
```

// --- MARK: - Added Functionality

```
func connectToDevice() -> Void {
    guard let peripheral = targetPeripheral else { return }
    central?.connect(peripheral, options: nil)
}
```

```
func disconnectFromDevice() -> Void {
    if let peripheral = targetPeripheral {
        central?.cancelPeripheralConnection(peripheral)
    }
}
```

```
func removeArrayData() -> Void {
    if let peripheral = targetPeripheral {
        central.cancelPeripheralConnection(peripheral)
    }
    rssiArray.removeAll()
    peripherals.removeAll()
}
```

```
func delayedConnection() -> Void {
    BlePeripheral.connectedPeripheral = targetPeripheral
    DispatchQueue.main.asyncAfter(deadline: .now() + 0.5, execute: {
        print("Delayed connection.")
    })
}
```

```

func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor service:
CIService, error: Error?) {
    guard let characteristics = service.characteristics else {
        return
    }
    print("Found \(characteristics.count) characteristics.")
    for characteristic in characteristics {
        if characteristic.uuid.isEqual(CBUUIDs.BLE_Characteristic_uuid_Rx) {
            rxCharacteristic = characteristic
            BlePeripheral.connectedRXChar = rxCharacteristic
            peripheral.setNotifyValue(true, for: rxCharacteristic)
            print("RX Characteristic: \(rxCharacteristic.uuid)")
        }
        if characteristic.uuid.isEqual(CBUUIDs.BLE_Characteristic_uuid_Tx){
            txCharacteristic = characteristic
            BlePeripheral.connectedTXChar = txCharacteristic
            print("TX Characteristic: \(txCharacteristic.uuid)")
        }
    }
    delayedConnection()
}
// MARK: - Data Handling
func peripheral(_ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic,
    error: Error?) {
    guard characteristic == rxCharacteristic,
        let data = characteristic.value,
        let receivedString = String(data: data, encoding: .utf8)
    else {
        print("didUpdateValueFor fired but guard failed — data: \(characteristic.value?.count ??
0) bytes")
        return
    }
    print("RAW \(data.count) bytes): \(receivedString.debugDescription)")

    let lines = receivedString.components(separatedBy: "\n")
    for line in lines {
        let trimmed = line.trimmingCharacters(in: .whitespacesAndNewlines)
        if trimmed.isEmpty { continue }
    }
}

```



```

// Expected format: p0, p1, p2, p3, imuX, imuY, imuZ, deviceTime
let parts = trimmed.components(separatedBy: ",").map { $0.trimmingCharacters(in:
.whitespaces) }
guard parts.count == 8,
    let p0 = Double(parts[0]),
    let p1 = Double(parts[1]),
    let p2 = Double(parts[2]),
    let p3 = Double(parts[3]),
    let imuX = Double(parts[4]),
    let imuY = Double(parts[5]),
    let imuZ = Double(parts[6]),
    let dt = Double(parts[7])
else {
    print("Malformed packet \(parts.count) fields): \(trimmed)")
    continue
}
print("p0:\(p0) p1:\(p1) p2:\(p2) p3:\(p3) | imu:\(imuX),\(imuY),\(imuZ) t:\(dt)")
DispatchQueue.main.async {
    let newPoint = BluetoothDataPoint(
        timestamp: Date(),
        pressure0: p0,
        pressure1: p1,
        pressure2: p2,
        pressure3: p3,
        imuX: imuX,
        imuY: imuY,
        imuZ: imuZ,
        deviceTime: dt
    )
    self.receivedDataHistory.append(newPoint)
    self.checkPressureAlerts(for: newPoint)
    self.saveHistory()
}
NotificationCenter.default.post(
    name: NSNotification.Name(rawValue: "Notify"),
    object: trimmed
)
}
}

```

```

func peripheral(_ peripheral: CBPeripheral, didReadRSSI RSSI: NSNumber, error: Error?) {
    peripheral.readRSSI()
}

func peripheral(_ peripheral: CBPeripheral, didWriteValueFor characteristic:
CBCharacteristic, error: Error?) {
    guard error == nil else {
        print("Error writing value: \(error!.localizedDescription)")
        return
    }
    print("Function: \(#function),Line: \(#line)")
    print("Message sent")
}

func peripheral(_ peripheral: CBPeripheral, didUpdateNotificationStateFor characteristic:
CBCharacteristic, error: Error?) {
    print("*****")
    print("Function: \(#function),Line: \(#line)")
    if (error != nil) {
        print("Error changing notification state:\(String(describing:
error?.localizedDescription))")
    } else {
        print("Characteristic's value subscribed")
    }
    if (characteristic.isNotifying) {
        print ("Subscribed. Notification has begun for: \(characteristic.uuid)")
    }
}

func centralManager(_ central: CBCentralManager, willRestoreState dict: [String : Any]) {
    // This is where the system passes back the peripheral it was monitoring.
    if let peripherals: [CBPeripheral] = dict[CBCentralManagerRestoredStatePeripheralsKey]
as? [CBPeripheral] {
        for peripheral in peripherals {
            // Re-assign the delegate so your manager can receive the disconnection event
            peripheral.delegate = self

            // Set your internal state properties if needed
            self.targetPeripheral = peripheral
            self.isConnected = (peripheral.state == .connected)
        }
    }
}

```

```
}
```

```
// MARK: - Hourly Aggregation (averages each pressure channel independently)
```

```
@objc private func aggregateAndSaveHourlyData() {  
    let now = Date()  
    let newPoints = receivedDataHistory.filter { $0.timestamp > self.lastAggregationTime }  
    guard !newPoints.isEmpty else {  
        print("Aggregation skipped: No new data points since \(self.lastAggregationTime).")  
        self.lastAggregationTime = now  
        return  
    }  
    let count = Double(newPoints.count)  
    // Average each sensor field independently  
    let avgP0 = newPoints.reduce(0.0) { $0 + $1.pressure0 } / count  
    let avgP1 = newPoints.reduce(0.0) { $0 + $1.pressure1 } / count  
    let avgP2 = newPoints.reduce(0.0) { $0 + $1.pressure2 } / count  
    let avgP3 = newPoints.reduce(0.0) { $0 + $1.pressure3 } / count  
    let avgImuX = newPoints.reduce(0.0) { $0 + $1.imuX } / count  
    let avgImuY = newPoints.reduce(0.0) { $0 + $1.imuY } / count  
    let avgImuZ = newPoints.reduce(0.0) { $0 + $1.imuZ } / count  
    // deviceTime: use the last reported value (not meaningful to average)  
    let lastDt = newPoints.last!.deviceTime  
    let averagedPoint = BluetoothDataPoint(  
        timestamp: now,  
        pressure0: avgP0,  
        pressure1: avgP1,  
        pressure2: avgP2,  
        pressure3: avgP3,  
        imuX: avgImuX,  
        imuY: avgImuY,  
        imuZ: avgImuZ,  
        deviceTime: lastDt  
    )  
    DispatchQueue.main.async {  
        let cutoff = self.lastAggregationTime  
        self.receivedDataHistory.removeAll { $0.timestamp > cutoff }  
        self.receivedDataHistory.append(averagedPoint)  
        self.lastAggregationTime = now  
        self.saveHistory()  
        print("Hourly Aggregation Complete:")  
    }  
}
```

```

        print(" - Processed \(newPoints.count) raw points.")
        print(" - Avg pressures: [\((String(format:"%.3f",avgP0)), \((String(format:"%.3f",avgP1)),
\((String(format:"%.3f",avgP2)), \((String(format:"%.3f",avgP3)))]")
        print(" - New History Count: \(self.receivedDataHistory.count)")
    }
}

```

// **MARK: - Retroactive 3-Hour Aggregation**

```

func retroactive3HourAggregation() {
    print("Starting retroactive 3-hour data aggregation...")
    let threeHoursAgo = Calendar.current.date(byAdding: .hour, value: -3, to: Date())!
    let dataToProcess = receivedDataHistory.filter { $0.timestamp < threeHoursAgo }
    let dataToKeep = receivedDataHistory.filter { $0.timestamp >= threeHoursAgo }
    guard !dataToProcess.isEmpty else {
        print("Retroactive aggregation skipped: No data older than 3 hours.")
        return
    }
    let calendar = Calendar.current
    var aggregatedPoints: [BluetoothDataPoint] = []
    // Sort and bucket into 3-hour windows (no channel grouping needed anymore)
    let sorted = dataToProcess.sorted { $0.timestamp < $1.timestamp }
    guard let firstTimestamp = sorted.first?.timestamp else { return }
    var windowStart = calendar.date(
        bySettingHour: (calendar.component(.hour, from: firstTimestamp) / 3) * 3,
        minute: 0, second: 0, of: firstTimestamp)!
    var windowEnd = calendar.date(byAdding: .hour, value: 3, to: windowStart)!
    var bucket: [BluetoothDataPoint] = []
    for point in sorted {
        if point.timestamp < windowEnd {
            bucket.append(point)
        } else {
            if let avg = averageBucket(bucket, timestamp: windowStart) {
                aggregatedPoints.append(avg)
            }
            bucket.removeAll()
            while point.timestamp >= windowEnd {
                windowStart = windowEnd
                windowEnd = calendar.date(byAdding: .hour, value: 3, to: windowStart)!
            }
            bucket.append(point)
        }
    }
}

```

```

    }
}
if let avg = averageBucket(bucket, timestamp: windowStart) {
    aggregatedPoints.append(avg)
}
DispatchQueue.main.async {
    self.receivedDataHistory = dataToKeep + aggregatedPoints
    self.saveHistory()
    print("Retroactive Aggregation Complete!")
    print(" - Processed \(dataToProcess.count) raw points → \(aggregatedPoints.count)
averaged points.")
    print(" - New Total History Count: \(self.receivedDataHistory.count)")
}
}

private func averageBucket(_ bucket: [BluetoothDataPoint], timestamp: Date) ->
BluetoothDataPoint? {
    guard !bucket.isEmpty else { return nil }
    let n = Double(bucket.count)
    return BluetoothDataPoint(
        timestamp: timestamp,
        pressure0: bucket.reduce(0.0) { $0 + $1.pressure0 } / n,
        pressure1: bucket.reduce(0.0) { $0 + $1.pressure1 } / n,
        pressure2: bucket.reduce(0.0) { $0 + $1.pressure2 } / n,
        pressure3: bucket.reduce(0.0) { $0 + $1.pressure3 } / n,
        imuX:      bucket.reduce(0.0) { $0 + $1.imuX }      / n,
        imuY:      bucket.reduce(0.0) { $0 + $1.imuY }      / n,
        imuZ:      bucket.reduce(0.0) { $0 + $1.imuZ }      / n,
        deviceTime: bucket.last!.deviceTime
    )
}
}
// BluetoothManager.swift
extension BluetoothManager {

}

```

//

```
// BluetoothView.swift
```

```
// LymphSense
```

//

```
import SwiftUI
```

```
import CoreBluetooth
```

import Combine

```
struct BluetoothView: View {
```

```
@ObservedObject var bluetoothManager: BluetoothManager
```

```
@State private var bluetoothStatusMessage: String = "Initializing Bluetooth..."
```

```
@State private var pulse = false
```

```
@State private var scanPulse = false
```

```
var body: some View {
```

ZStack {

```
Color(red: 0.07, green: 0.07, blue: 0.1).ignoresSafeArea()
```

RadialGradient(

```
colors: [Color(red: 0.15, green: 0.25, blue: 0.45).opacity(0.4), .clear],
```

center: .top, startRadius: 0, endRadius: 400

)

`.ignoresSafeArea()`

$$\text{VStack}(\text{spacing: } 0) \{$$

```
// MARK: Header
```

HStack {

```
VStack(alignment: .leading, spacing: 4) {
```

Text("Connect")

```
.font(.system(size: 28, weight: .bold, design: .rounded))
```

```
.foregroundColor(.white)
```

Text(bluetoothStatusMessage)

```
.font(.system(size: 13, weight: .regular))
```

```
.foregroundColor(.white.opacity(0.4))
```

```
.tracking(1.0)
```

}

Spacer()

```
// Connection pill
```

HStack(spacing: 6) {

ZStack {

```
if bluetoothManager.isConnected {
```

Circle()

```
.fill(Color(red: 0.035, green: 0.612, blue: 0.063))
```

```

        .frame(width: 8, height: 8)
        .scaleEffect(pulse ? 1.4 : 1.0)
        .opacity(pulse ? 0.3 : 1.0)
        .animation(.easeInOut(duration: 1.2).repeatForever(), value: pulse)
    }
    Circle()
        .fill(blueetoothManager.isConnected
            ? Color(red: 0.035, green: 0.612, blue: 0.063)
            : Color(red: 0.839, green: 0.035, blue: 0.035))
        .frame(width: 8, height: 8)
    }
    Text(blueetoothManager.isConnected ? "Connected" : "Disconnected")
        .font(.system(size: 12, weight: .semibold))
        .foregroundColor(.white.opacity(0.85))
    }
    .padding(.horizontal, 12)
    .padding(.vertical, 8)
    .background(Color.white.opacity(0.07))
    .clipShape(Capsule())
    .overlay(Capsule().strokeBorder(Color.white.opacity(0.1), lineWidth: 1))
}
.padding(.horizontal, 24)
.padding(.top, 20)
.padding(.bottom, 24)
// MARK: Scan Button
Button {
    bluetoothManager.startScan()
} label: {
    HStack(spacing: 10) {
        ZStack {
            if bluetoothManager.isScanning {
                Circle()
                    .fill(Color(red: 0.3, green: 0.6, blue: 1.0))
                    .frame(width: 10, height: 10)
                    .scaleEffect(scanPulse ? 2.0 : 1.0)
                    .opacity(scanPulse ? 0.0 : 0.6)
                    .animation(.easeOut(duration: 1.0).repeatForever(autoreverses: false),
value: scanPulse)
            }
            Circle()

```

```

        .fill(blueetoothManager.isScanning
            ? Color(red: 0.3, green: 0.6, blue: 1.0)
            : Color.white.opacity(0.4))
        .frame(width: 10, height: 10)
    }
    Text(blueetoothManager.isScanning ? "Scanning..." : "Scan for Devices")
        .font(.system(size: 15, weight: .semibold))
        .foregroundColor(.white)
    }
    .frame(maxWidth: .infinity)
    .padding(.vertical, 16)
    .background(
        RoundedRectangle(cornerRadius: 14)
        .fill(blueetoothManager.isScanning
            ? Color(red: 0.3, green: 0.6, blue: 1.0).opacity(0.15)
            : Color.white.opacity(0.07))
        .overlay(
            RoundedRectangle(cornerRadius: 14)
            .strokeBorder(
                blueetoothManager.isScanning
                    ? Color(red: 0.3, green: 0.6, blue: 1.0).opacity(0.4)
                    : Color.white.opacity(0.1),
                lineWidth: 1
            )
        )
    )
}
.disabled(blueetoothManager.isScanning)
.padding(.horizontal, 16)
.padding(.bottom, 20)
// MARK: Devices Section
HStack {
    Text("Nearby Devices")
        .font(.system(size: 11, weight: .semibold))
        .foregroundColor(.white.opacity(0.35))
        .tracking(1.5)
        .textCase(.uppercase)
    Spacer()
    if !blueetoothManager.peripherals.isEmpty {
        Text("\${blueetoothManager.peripherals.count} found")
    }
}

```



```

        .font(.system(size: 11, weight: .medium))
        .foregroundColor(.white.opacity(0.2))
    }
}
.padding(.horizontal, 28)
.padding(.bottom, 8)
if bluetoothManager.peripherals.isEmpty {
    // Empty state
    VStack(spacing: 12) {
        Image(systemName: "antenna.radiowaves.left.and.right")
            .font(.system(size: 32))
            .foregroundColor(.white.opacity(0.12))
        Text(bluetoothManager.isScanning ? "Looking for devices..." : "No devices
found.\nTap Scan to search.")
            .font(.system(size: 14))
            .foregroundColor(.white.opacity(0.28))
            .multilineTextAlignment(.center)
    }
    .frame(maxWidth: .infinity)
    .padding(.vertical, 40)
    .background(
        RoundedRectangle(cornerRadius: 16)
            .fill(Color.white.opacity(0.03))
            .overlay(
                RoundedRectangle(cornerRadius: 16)
                    .strokeBorder(Color.white.opacity(0.06), lineWidth: 1)
            )
    )
    .padding(.horizontal, 16)
} else {
    ScrollView(showsIndicators: false) {
        LazyVStack(spacing: 6) {
            ForEach(bluetoothManager.peripherals, id: \.identifier) { peripheral in
                peripheralRow(peripheral)
            }
        }
        .padding(.horizontal, 16)
        .padding(.bottom, 24)
    }
}
}

```

```

        Spacer()
    }
}
.onAppear {
    pulse = true
    scanPulse = true
    updateStatusMessage(for: bluetoothManager.central.state)
}
.onReceive(bluetoothManager.objectWillChange) {
    updateStatusMessage(for: bluetoothManager.central.state)
}
}
// MARK: - Peripheral Row
@ViewBuilder
private func peripheralRow(_ peripheral: CBPeripheral) -> some View {
    let isConnected = peripheral.state == .connected
    let isConnecting = peripheral.state == .connecting
    let accentColor: Color = isConnected
        ? Color(red: 0.035, green: 0.612, blue: 0.063)
        : Color(red: 0.3, green: 0.6, blue: 1.0)
    Button {
        guard peripheral.state == .disconnected else { return }
        bluetoothManager.connect(peripheral)
    } label: {
        HStack(spacing: 14) {
            // Device icon badge
            ZStack {
                RoundedRectangle(cornerRadius: 8)
                    .fill(accentColor.opacity(0.12))
                    .frame(width: 36, height: 36)
                Image(systemName: isConnected ? "checkmark.circle.fill" :
"sensor.tag.radiowaves.forward.fill")
                    .font(.system(size: 16))
                    .foregroundColor(accentColor)
            }
            VStack(alignment: .leading, spacing: 2) {
                Text("LymphSense Device" ?? "Unknown Device")
                    .font(.system(size: 15, weight: .semibold))
                    .foregroundColor(.white)
                Text(peripheral.identifier.uuidString)
            }
        }
    }
}

```

```

        .font(.system(size: 11, design: .monospaced))
        .foregroundColor(.white.opacity(0.3))
        .lineLimit(1)
        .truncationMode(.middle)
    }
    Spacer()
    if isConnected {
        ProgressView()
        .tint(.white.opacity(0.4))
        .scaleEffect(0.8)
    } else if isConnected {
        Text("Connected")
        .font(.system(size: 11, weight: .semibold))
        .foregroundColor(Color(red: 0.035, green: 0.612, blue: 0.063))
        .padding(.horizontal, 8)
        .padding(.vertical, 4)
        .background(Color(red: 0.035, green: 0.612, blue: 0.063).opacity(0.12))
        .clipShape(Capsule())
    } else {
        Image(systemName: "chevron.right")
        .font(.system(size: 12, weight: .semibold))
        .foregroundColor(.white.opacity(0.25))
    }
}
.padding(.horizontal, 16)
.padding(.vertical, 14)
.background(
    RoundedRectangle(cornerRadius: 14)
    .fill(Color.white.opacity(0.05))
    .overlay(
        RoundedRectangle(cornerRadius: 14)
        .strokeBorder(accentColor.opacity(isConnected ? 0.3 : 0.1), lineWidth: 1)
    )
)
}
.buttonStyle(.plain)
}
// MARK: - Status Helper
private func updateStatusMessage(for state: CBManagerState) {
    switch state {

```

```

        case .poweredOn:
            bluetoothStatusMessage = bluetoothManager.isScanning ? "Scanning for devices..." :
"Ready to scan"
        case .poweredOff:
            bluetoothStatusMessage = "Bluetooth is off"
        case .unauthorized:
            bluetoothStatusMessage = "Bluetooth access denied"
        case .unsupported:
            bluetoothStatusMessage = "Bluetooth LE not supported"
        case .resetting:
            bluetoothStatusMessage = "Bluetooth resetting..."
        case .unknown:
            bluetoothStatusMessage = "Initializing..."
        @unknown default:
            bluetoothStatusMessage = "Unknown state"
    }
}
}

```

Section IV. CBUUIDs.swift

```

//
// CBUUIDs.swift
// LymphSense
//
import Foundation
import CoreBluetooth
struct CBUUIDs{
    static let kBLEService_UUID = "6e400001-b5a3-f393-e0a9-e50e24dcca9e"
    static let kBLE_Characteristic_uuid_Tx = "6e400002-b5a3-f393-e0a9-e50e24dcca9e"
    static let kBLE_Characteristic_uuid_Rx = "6e400003-b5a3-f393-e0a9-e50e24dcca9e"
    static let MaxCharacters = 20
    static let BLEService_UUID = CBUUID(string: kBLEService_UUID)
    static let BLE_Characteristic_uuid_Tx = CBUUID(string:
kBLE_Characteristic_uuid_Tx)//(Property = Write without response)
    static let BLE_Characteristic_uuid_Rx = CBUUID(string: kBLE_Characteristic_uuid_Rx)//
(Property = Read/Notify)
}

```

Section V. NotificationManager.swift

```

//
// NotificationManager.swift

```

```

// LymphSense
//
import Foundation
import UserNotifications
import UIKit
class NotificationManager: NSObject, UNUserNotificationCenterDelegate {

    static let shared = NotificationManager()

    private override init() {
        super.init()
        UNUserNotificationCenter.current().delegate = self
    }

    // MARK: - App State

    private var isActive: Bool {
        UIApplication.shared.applicationState == .active
    }

    // MARK: - Delegate (Foreground behavior)

    func userNotificationCenter(_ center: UNUserNotificationCenter,
                               willPresent notification: UNNotification,
                               withCompletionHandler completionHandler: @escaping
(UnNotificationPresentationOptions) -> Void) {

        // Show banner even when app is open
        completionHandler([.banner, .sound, .badge])
    }

    func userNotificationCenter(_ center: UNUserNotificationCenter,
                               didReceive response: UNNotificationResponse,
                               withCompletionHandler completionHandler: @escaping () -> Void) {

        print("Notification tapped by user")
        completionHandler()
    }

    // MARK: - Permission Request

```

```

func requestAuthorization() {
    UNNotificationCenter.current().requestAuthorization(options: [.alert, .sound, .badge])
{ granted, error in

    if let error = error {
        print("Notification permission error: \(error)")
    }

    print("Notification permission granted: \(granted)")

    DispatchQueue.main.async {
        UIApplication.shared.registerForRemoteNotifications()
    }
}
}

```

// **MARK: - CORE SAFE NOTIFICATION ENGINE**

```

private func sendNotification(title: String,
                             body: String,
                             identifier: String = UUID().uuidString) {

    // Only send system notification if app is NOT active
    // (prevents in-app spam + ensures correct UX behavior)

    guard !isActive else {
        print("App is active → skipping system notification (use UI alert instead)")
        return
    }

    let content = UNMutableNotificationContent()
    content.title = title
    content.body = body
    content.sound = .default

    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 1, repeats: false)

    let request = UNNotificationRequest(
        identifier: identifier,

```

```

        content: content,
        trigger: trigger
    )

    UNUserNotificationCenter.current().add(request) { error in
        if let error = error {
            print("Failed to schedule notification: \(error.localizedDescription)")
        }
    }
}

// MARK: - PUBLIC API

func scheduleLocalNotification() {
    sendNotification(
        title: "LymphSense",
        body: "Test notification"
    )
}

func schedulePressureAlert(zones: String) {
    sendNotification(
        title: "LymphSense",
        body: "\(zones) pressure is outside safe range. Please readjust bandaging."
    )
}

func scheduleDisconnectionNotification(deviceName: String?) {
    let name = "LymphSense Device"

    sendNotification(
        title: "LymphSense",
        body: "Connection lost. Please reconnect \(name) to continue monitoring."
    )
}
}

Section VI. ArmPressureView.swift
//
// ArmPressureView.swift
// LymphSense

```

```
//
import SwiftUI
struct ArmPressureView: View {
    let values: [Double?]
    let isConnected: Bool
    // Specs keyed 0–3 to match the rest of the updated codebase
    private struct SensorSpec {
        let low: Double
        let high: Double
        let tolerance: Double
    }
    private let sensorSpecs: [Int: SensorSpec] = [
        0: SensorSpec(low: 18, high: 28, tolerance: 2), // Upper Arm    mmHg
        1: SensorSpec(low: 35, high: 45, tolerance: 2), // Upper Middle mmHg
        2: SensorSpec(low: 36, high: 46, tolerance: 2), // Lower Middle mmHg
        3: SensorSpec(low: 40, high: 50, tolerance: 2), // Lower Arm    mmHg
    ]
    // Returns (r, g, b) components in 0–1 range for a given sensor index
    private func rgb(for value: Double?, sensor: Int) -> (Double, Double, Double) {
        guard let value, let spec = sensorSpecs[sensor] else {
            return (0.839, 0.035, 0.035) // no reading → red
        }
        if value >= spec.low && value <= spec.high {
            return (0.035, 0.612, 0.063) // in range → green
        } else if value < spec.low {
            let delta = spec.low - value
            return delta <= spec.tolerance
                ? (0.906, 0.906, 0.078) // borderline low → yellow
                : (0.839, 0.035, 0.035) // too low → red
        } else {
            let delta = value - spec.high
            return delta <= spec.tolerance
                ? (0.906, 0.906, 0.078) // borderline high → yellow
                : (0.839, 0.035, 0.035) // over range → red
        }
    }
    // Blend two (r,g,b) tuples by t in 0–1
    private func blendRGB(
        _ a: (Double, Double, Double),
        _ b: (Double, Double, Double),

```



```

    t: Double
) -> Color {
    Color(
        red:  a.0 + (b.0 - a.0) * t,
        green: a.1 + (b.1 - a.1) * t,
        blue:  a.2 + (b.2 - a.2) * t
    )
}
private func color(_ rgb: (Double, Double, Double)) -> Color {
    Color(red: rgb.0, green: rgb.1, blue: rgb.2)
}
private var armGradient: LinearGradient {
    let c0 = rgb(for: values.count > 0 ? values[0] : nil, sensor: 0)
    let c1 = rgb(for: values.count > 1 ? values[1] : nil, sensor: 1)
    let c2 = rgb(for: values.count > 2 ? values[2] : nil, sensor: 2)
    let c3 = rgb(for: values.count > 3 ? values[3] : nil, sensor: 3)
    return LinearGradient(
        stops: [
            .init(color: color(c0),          location: 0.00),
            .init(color: color(c0),          location: 0.22),
            .init(color: blendRGB(c0, c1, t: 0.5), location: 0.25),
            .init(color: color(c1),          location: 0.27),
            .init(color: color(c1),          location: 0.47),
            .init(color: blendRGB(c1, c2, t: 0.5), location: 0.50),
            .init(color: color(c2),          location: 0.52),
            .init(color: color(c2),          location: 0.72),
            .init(color: blendRGB(c2, c3, t: 0.5), location: 0.75),
            .init(color: color(c3),          location: 0.77),
            .init(color: color(c3),          location: 1.00),
        ],
        startPoint: .leading,
        endPoint:   .trailing
    )
}
var body: some View {
    GeometryReader { geo in
        let w = geo.size.width
        let h = geo.size.height
        // Guard against zero/invalid dimensions on first render pass
        guard w > 0, h > 0 else { return AnyView(Color.clear) }
    }
}

```

```

return AnyView(
  ZStack {
    armGradient
      .mask(
        Image("arm")
          .resizable()
          .scaledToFit()
          .frame(width: h, height: w)
          .rotationEffect(.degrees(-90))
          .frame(width: w, height: h)
      )
    Image("arm")
      .resizable()
      .scaledToFit()
      .frame(width: h, height: w)
      .rotationEffect(.degrees(-90))
      .frame(width: w, height: h)
      .blendMode(.multiply)
  }
)
}
}
}

```

Section VII. ContentView.swift

```
//
// ContentView.swift
// LymphSense
//
import SwiftUI
struct ContentView: View {
    // Bluetooth Manager is the source of truth for the connection state.
    // Ensure this is the highest level where the manager is instantiated.
    @StateObject var bluetoothManager = BluetoothManager()
    // Local state derived from the manager's published property.
    @State private var isAuthenticatedAndConnected: Bool = false
    var body: some View {
        // The NavigationStack is the root container for navigation context.
        NavigationStack {
            MainTabView()
                .environmentObject(bluetoothManager) // Pass the manager down
        }
        .environmentObject(bluetoothManager)

        // Observe the manager's connection status.
        .onReceive(bluetoothManager.$isConnected) { isConnected in
            // Update the local state, which causes the body to redraw and switch views.
            isAuthenticatedAndConnected = isConnected

            if isConnected {
                print("Device Connected. Root view switched to MainTabView/HomeView.")
            } else {
                print("Device Disconnected. Root view switched to BluetoothView.")
            }
        }
    }
}
```

Section VIII. DataView.swift

```
//
// DataView.swift
// LymphSense
//
import SwiftUI
struct DataView: View {
    @EnvironmentObject var bluetoothManager: BluetoothManager
    @State private var pulse = false
    private static var dateFormatter: DateFormatter = {
        let f = DateFormatter()
        f.dateStyle = .none
        f.timeStyle = .medium
        return f
    }()
    // Color-codes a pressure reading relative to a neutral baseline
    private func pressureColor(for value: Double) -> Color {
        let diff = abs(value - 0.9)
        if diff < 0.1 { return Color(red: 0.035, green: 0.612, blue: 0.063) }
        if diff < 0.2 { return Color(red: 0.906, green: 0.906, blue: 0.078) }
        return Color(red: 0.839, green: 0.035, blue: 0.035)
    }
    var body: some View {
        ZStack {
            Color(red: 0.07, green: 0.07, blue: 0.1).ignoresSafeArea()
            RadialGradient(
                colors: [Color(red: 0.15, green: 0.25, blue: 0.45).opacity(0.4), .clear],
                center: .top, startRadius: 0, endRadius: 400
            )
            .ignoresSafeArea()
            VStack(spacing: 0) {
                // MARK: Header
                HStack {
                    VStack(alignment: .leading, spacing: 4) {
                        Text("Data Log")
                            .font(.system(size: 28, weight: .bold, design: .rounded))
                            .foregroundColor(.white)
                        Text("\(bluetoothManager.receivedDataHistory.count) Entries Recorded")
                            .font(.system(size: 13, weight: .regular))
                            .foregroundColor(.white.opacity(0.4))
                    }
                }
            }
        }
    }
}
```

```

        .tracking(1.2)
    }
    Spacer()
    // Connection pill
    HStack(spacing: 6) {
        ZStack {
            if bluetoothManager.isConnected {
                Circle()
                    .fill(Color(red: 0.035, green: 0.612, blue: 0.063))
                    .frame(width: 8, height: 8)
                    .scaleEffect(pulse ? 1.4 : 1.0)
                    .opacity(pulse ? 0.3 : 1.0)
                    .animation(.easeInOut(duration: 1.2).repeatForever(), value: pulse)
            }
            Circle()
                .fill(bluetoothManager.isConnected
                    ? Color(red: 0.035, green: 0.612, blue: 0.063)
                    : Color(red: 0.839, green: 0.035, blue: 0.035))
                .frame(width: 8, height: 8)
        }
        Text(bluetoothManager.isConnected ? "Connected" : "Disconnected")
            .font(.system(size: 12, weight: .semibold))
            .foregroundColor(.white.opacity(0.85))
    }
    .padding(.horizontal, 12)
    .padding(.vertical, 8)
    .background(Color.white.opacity(0.07))
    .clipShape(Capsule())
    .overlay(Capsule().strokeBorder(Color.white.opacity(0.1), lineWidth: 1))
}
.padding(.horizontal, 24)
.padding(.top, 20)
.padding(.bottom, 16)

// MARK: Column Headers
HStack(spacing: 0) {
    Text("Time")
        .frame(width: 72, alignment: .leading)
    Text("P0")
        .frame(maxWidth: .infinity, alignment: .center)
}

```

```

Text("P1")
    .frame(maxWidth: .infinity, alignment: .center)
Text("P2")
    .frame(maxWidth: .infinity, alignment: .center)
Text("P3")
    .frame(maxWidth: .infinity, alignment: .center)
Text("Ax")
    .frame(maxWidth: .infinity, alignment: .center)
Text("Ay")
    .frame(maxWidth: .infinity, alignment: .center)
Text("Az")
    .frame(maxWidth: .infinity, alignment: .center)
}
.font(system(size: 11, weight: .semibold))
.foregroundColor(.white.opacity(0.3))
.tracking(1.2)
.textCase(.uppercase)
.padding(.horizontal, 20)
.padding(.bottom, 8)

// MARK: List
if bluetoothManager.receivedDataHistory.isEmpty {
    Spacer()
    VStack(spacing: 12) {
        Image(systemName: "list.bullet.rectangle")
            .font(.system(size: 32))
            .foregroundColor(.white.opacity(0.15))
        Text("No data recorded yet.")
            .font(.system(size: 14))
            .foregroundColor(.white.opacity(0.3))
    }
    Spacer()
} else {
    ScrollView(showsIndicators: false) {
        LazyVStack(spacing: 6) {
            ForEach(bluetoothManager.receivedDataHistory.reversed()) { dataPoint in
                dataRow(dataPoint)
            }
        }
    }
    .padding(.horizontal, 16)
}

```

```

        .padding(.bottom, 24)
    }
}
}
}
.onAppear { pulse = true }
}

```

// **MARK: - Data Row**

@ViewBuilder

```

private func dataRow(_ dp: BluetoothDataPoint) -> some View {
    VStack(spacing: 6) {
        // Row 1: timestamp + device time
        HStack {
            Text(dp.timestamp, formatter: DataView.dateFormatter)
                .font(.system(size: 11, design: .monospaced))
                .foregroundColor(.white.opacity(0.4))
            Spacer()
            Text("t = \(String(format: "%.2f", dp.deviceTime)) ms")
                .font(.system(size: 11, design: .monospaced))
                .foregroundColor(.white.opacity(0.25))
        }
        // Row 2: pressure sensors
        HStack(spacing: 0) {
            ForEach([
                ("P0", dp.pressure0),
                ("P1", dp.pressure1),
                ("P2", dp.pressure2),
                ("P3", dp.pressure3)
            ], id: \.0) { label, val in
                let color = pressureColor(for: val)
                VStack(spacing: 2) {
                    Text(label)
                        .font(.system(size: 9, weight: .semibold))
                        .foregroundColor(.white.opacity(0.3))
                        .tracking(0.8)
                    Text(String(format: "%.3f", val))
                        .font(.system(size: 12, weight: .bold, design: .monospaced))
                        .foregroundColor(color)
                }
            }
        }
    }
}

```

```

        .frame(maxWidth: .infinity)
        .padding(.vertical, 4)
        .background(color.opacity(0.07))
        .cornerRadius(6)
    }
}
// Row 3: IMU axes
HStack(spacing: 0) {
    ForEach([
        ("Ax", dp.imuX),
        ("Ay", dp.imuY),
        ("Az", dp.imuZ)
    ], id: \.0) { label, val in
        VStack(spacing: 2) {
            Text(label)
                .font(.system(size: 9, weight: .semibold))
                .foregroundColor(.white.opacity(0.3))
                .tracking(0.8)
            Text(String(format: "%.3f", val))
                .font(.system(size: 12, weight: .bold, design: .monospaced))
                .foregroundColor(Color(red: 0.5, green: 0.75, blue: 1.0))
        }
        .frame(maxWidth: .infinity)
        .padding(.vertical, 4)
        .background(Color(red: 0.3, green: 0.6, blue: 1.0).opacity(0.07))
        .cornerRadius(6)
    }
}
}
.padding(.horizontal, 14)
.padding(.vertical, 10)
.background(
    RoundedRectangle(cornerRadius: 14)
        .fill(Color.white.opacity(0.04))
        .overlay(
            RoundedRectangle(cornerRadius: 14)
                .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
        )
)
}

```



```
}  
  
struct DataView_Previews: PreviewProvider {  
  static var previews: some View {  
    DataView()  
      .environmentObject(BluetoothManager())  
  }  
}
```

Section IX. TrendsView.swift

```
//  
// TrendsView.swift  
// LymphSense  
//  
import SwiftUI  
struct TrendsView: View {  
    @EnvironmentObject var bluetoothManager: BluetoothManager  
    @State private var selectedDate = Date()  
    @State private var pulse = false  
    @State private var calendarExpanded = false  
    private static var timeFormatter: DateFormatter = {  
        let f = DateFormatter()  
        f.dateStyle = .none  
        f.timeStyle = .medium  
        return f  
    }()  
    // MARK: - Data access  
    private var activeData: [BluetoothDataPoint] { bluetoothManager.receivedDataHistory }  
    private var filteredData: [BluetoothDataPoint] {  
        activeData  
        .filter { Calendar.current.isDate($0.timestamp, inSameDayAs: selectedDate) }  
        .sorted { $0.timestamp > $1.timestamp } // newest first  
    }  
    // Per-sensor daily high/low across all 4 pressure fields combined  
    private var dailyHigh: Double? {  
        filteredData.flatMap { [$0.pressure0, $0.pressure1, $0.pressure2, $0.pressure3] }.max()  
    }  
    private var dailyLow: Double? {  
        filteredData.flatMap { [$0.pressure0, $0.pressure1, $0.pressure2, $0.pressure3] }.min()  
    }  
    /// Latest pressure reading for sensor 0–3 from the most recent entry on selectedDate  
    private func latestValue(for sensor: Int) -> Double? {  
        guard let latest = filteredData.first else { return nil }  
        switch sensor {  
        case 0: return latest.pressure0  
        case 1: return latest.pressure1  
        case 2: return latest.pressure2  
        case 3: return latest.pressure3  
        default: return nil
```

```

    }
  }
  private let sensorOrder: [Int] = [0, 1, 2, 3]
  private let sensorLabels: [String] = ["Upper Arm", "Upper Middle", "Lower Middle", "Lower
Arm"]
  // MARK: - Body
  var body: some View {
    ZStack {
      Color(red: 0.07, green: 0.07, blue: 0.1).ignoresSafeArea()
      RadialGradient(
        colors: [Color(red: 0.15, green: 0.25, blue: 0.45).opacity(0.4), .clear],
        center: .top, startRadius: 0, endRadius: 400
      )
      .ignoresSafeArea()
      ScrollView(showsIndicators: false) {
        VStack(spacing: 0) {
          // MARK: Header
          HStack {
            VStack(alignment: .leading, spacing: 4) {
              Text("Trends")
                .font(.system(size: 28, weight: .bold, design: .rounded))
                .foregroundColor(.white)
              Text("Pressure History")
                .font(.system(size: 13, weight: .regular))
                .foregroundColor(.white.opacity(0.4))
                .tracking(1.5)
                .textCase(.uppercase)
            }
            Spacer()
            HStack(spacing: 6) {
              ZStack {
                if bluetoothManager.isConnected {
                  Circle()
                    .fill(Color(red: 0.035, green: 0.612, blue: 0.063))
                    .frame(width: 8, height: 8)
                    .scaleEffect(pulse ? 1.4 : 1.0)
                    .opacity(pulse ? 0.3 : 1.0)
                    .animation(.easeInOut(duration: 1.2).repeatForever(), value: pulse)
                }
                Circle()
              }
            }
          }
        }
      }
    }
  }

```

```

        .fill(blueetoothManager.isConnected
            ? Color(red: 0.035, green: 0.612, blue: 0.063)
            : Color(red: 0.839, green: 0.035, blue: 0.035))
        .frame(width: 8, height: 8)
    }
    Text(blueetoothManager.isConnected ? "Connected" : "Disconnected")
        .font(.system(size: 12, weight: .semibold))
        .foregroundColor(.white.opacity(0.85))
    }
    .padding(.horizontal, 12)
    .padding(.vertical, 8)
    .background(Color.white.opacity(0.07))
    .clipShape(Capsule())
    .overlay(Capsule().strokeBorder(Color.white.opacity(0.1), lineWidth: 1))
}
.padding(.horizontal, 24)
.padding(.top, 20)
.padding(.bottom, 24)
// MARK: Calendar
sectionHeader("Select Date")
VStack(spacing: 0) {
    // Tappable date pill
    Button {
        withAnimation(.spring(duration: 0.35)) {
            calendarExpanded.toggle()
        }
    } label: {
        HStack(spacing: 12) {
            ZStack {
                RoundedRectangle(cornerRadius: 8)
                    .fill(Color(red: 0.3, green: 0.6, blue: 1.0).opacity(0.12))
                    .frame(width: 36, height: 36)
                Image(systemName: "calendar")
                    .font(.system(size: 15))
                    .foregroundColor(Color(red: 0.3, green: 0.6, blue: 1.0))
            }
            VStack(alignment: .leading, spacing: 2) {
                Text(selectedDate.formatted(.dateTime.weekday(.wide).month(.wide).day().year()))
                    .font(.system(size: 15, weight: .semibold))
            }
        }
    }
}

```

```

        .foregroundColor(.white)
        Text(calendarExpanded ? "Tap to collapse" : "Tap to change date")
        .font(.system(size: 12))
        .foregroundColor(.white.opacity(0.4))
    }
    Spacer()
    Image(systemName: "chevron.down")
        .font(.system(size: 12, weight: .semibold))
        .foregroundColor(.white.opacity(0.3))
        .rotationEffect(.degrees(calendarExpanded ? 180 : 0))
        .animation(.spring(duration: 0.35), value: calendarExpanded)
    }
    .padding(.horizontal, 16)
    .padding(.vertical, 12)
}
.buttonStyle(.plain)
if calendarExpanded {
    Rectangle()
        .fill(Color.white.opacity(0.06))
        .frame(height: 1)
        .padding(.horizontal, 16)
    DatePicker("Select a Date", selection: $selectedDate, displayedComponents:
.date)

        .datePickerStyle(.graphical)
        .labelsHidden()
        .tint(Color(red: 0.3, green: 0.6, blue: 1.0))
        .colorScheme(.dark)
        .padding(12)
        .transition(.opacity.combined(with: .move(edge: .top)))
        .onChange(of: selectedDate) {
            withAnimation(.spring(duration: 0.35)) {
                calendarExpanded = false
            }
        }
    }
}
}
.background(
    RoundedRectangle(cornerRadius: 16)
        .fill(Color.white.opacity(0.05))
        .overlay(

```

```

        RoundedRectangle(cornerRadius: 16)
            .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
        )
    )
    .padding(.horizontal, 16)
    .padding(.bottom, 20)
    .animation(.spring(duration: 0.35), value: calendarExpanded)
    // MARK: Daily Summary
    sectionHeader("Summary —
\\(selectedDate.formatted(.dateTime.month(.abbreviated).day().year()))")
    if filteredData.isEmpty {
        HStack(spacing: 12) {
            Image(systemName: "chart.xyaxis.line")
                .font(.system(size: 20))
                .foregroundColor(.white.opacity(0.2))
            Text("No data recorded on this date.")
                .font(.system(size: 14))
                .foregroundColor(.white.opacity(0.35))
        }
        .frame(maxWidth: .infinity)
        .padding(.vertical, 28)
        .background(
            RoundedRectangle(cornerRadius: 16)
                .fill(Color.white.opacity(0.04))
                .overlay(
                    RoundedRectangle(cornerRadius: 16)
                        .strokeBorder(Color.white.opacity(0.07), lineWidth: 1)
                )
        )
    )
    .padding(.horizontal, 16)
    .padding(.bottom, 20)
} else {
    // MARK: High / Low (across all sensors)
    if let high = dailyHigh, let low = dailyLow {
        HStack(spacing: 10) {
            statCard(
                icon: "arrow.up.circle.fill",
                label: "Daily High",
                value: high,
                color: Color(red: 0.839, green: 0.035, blue: 0.035)
            )

```

```

    )
    statCard(
      icon: "arrow.down.circle.fill",
      label: "Daily Low",
      value: low,
      color: Color(red: 0.3, green: 0.6, blue: 1.0)
    )
  }
  .padding(.horizontal, 16)
  .padding(.bottom, 20)
}
// MARK: Per-sensor latest readings
sectionHeader("Latest by Zone")
VStack(spacing: 1) {
  ForEach(Array(zip(sensorOrder, sensorLabels)), id: \.0) { sensor, label in
    if let val = latestValue(for: sensor) {
      if sensor != 0 { divider() }
      sensorRow(sensor: sensor, label: label, value: val)
    }
  }
}
.background(
  RoundedRectangle(cornerRadius: 16)
    .fill(Color.white.opacity(0.05))
    .overlay(
      RoundedRectangle(cornerRadius: 16)
        .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
    )
)
.padding(.horizontal, 16)
.padding(.bottom, 20)
// MARK: Earliest / Latest entry timestamps
sectionHeader("Measurements")
VStack(spacing: 1) {
  if let latest = filteredData.first {
    measurementRow(label: "Latest", entry: latest)
  }
  if filteredData.count > 1, let earliest = filteredData.last {
    divider()
    measurementRow(label: "Earliest", entry: earliest)
  }
}

```

```

        }
    }
    .background(
        RoundedRectangle(cornerRadius: 16)
        .fill(Color.white.opacity(0.05))
        .overlay(
            RoundedRectangle(cornerRadius: 16)
            .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
        )
    )
    .padding(.horizontal, 16)
    .padding(.bottom, 40)
}
}
}
}
.onAppear { pulse = true }
}
// MARK: - Subviews
@ViewBuilder
private func sectionHeader(_ title: String) -> some View {
    HStack {
        Text(title)
        .font(.system(size: 11, weight: .semibold))
        .foregroundColor(.white.opacity(0.35))
        .tracking(1.5)
        .textCase(.uppercase)
        Spacer()
    }
    .padding(.horizontal, 28)
    .padding(.bottom, 8)
}
@ViewBuilder
private func statCard(icon: String, label: String, value: Double, color: Color) -> some View {
    VStack(alignment: .leading, spacing: 10) {
        HStack(spacing: 8) {
            Image(systemName: icon)
                .font(.system(size: 16))
                .foregroundColor(color)
            Text(label)

```



```

        .font(.system(size: 12, weight: .medium))
        .foregroundColor(.white.opacity(0.45))
        .tracking(0.5)
    }
    Text(String(format: "%.1f mmHg", value))
        .font(.system(size: 22, weight: .bold, design: .monospaced))
        .foregroundColor(color)
    }
    .frame(maxWidth: .infinity, alignment: .leading)
    .padding(16)
    .background(
        RoundedRectangle(cornerRadius: 16)
        .fill(Color.white.opacity(0.05))
        .overlay(
            RoundedRectangle(cornerRadius: 16)
            .strokeBorder(color.opacity(0.2), lineWidth: 1)
        )
    )
}
@ViewBuilder
private func sensorRow(sensor: Int, label: String, value: Double) -> some View {
    let status = pressureStatus(for: value, sensor: sensor)
    HStack(spacing: 14) {
        ZStack {
            RoundedRectangle(cornerRadius: 8)
                .fill(status.color.opacity(0.12))
                .frame(width: 36, height: 36)
            Image(systemName: "waveform.path.ecg")
                .font(.system(size: 15))
                .foregroundColor(status.color)
        }
        VStack(alignment: .leading, spacing: 2) {
            Text(label)
                .font(.system(size: 15, weight: .semibold))
                .foregroundColor(.white)
            Text("Sensor \(sensor) — \(status.label)")
                .font(.system(size: 12))
                .foregroundColor(status.color.opacity(0.8))
        }
        Spacer()
    }
}

```

```

        Text(String(format: "%.1f mmHg", value))
            .font(.system(size: 15, weight: .bold, design: .monospaced))
            .foregroundColor(status.color)
    }
    .padding(.horizontal, 16)
    .padding(.vertical, 12)
}
/// Shows timestamp + all 4 pressure readings + IMU for an entry
@ViewBuilder
private func measurementRow(label: String, entry: BluetoothDataPoint) -> some View {
    HStack(spacing: 14) {
        ZStack {
            RoundedRectangle(cornerRadius: 8)
                .fill(Color(red: 0.3, green: 0.6, blue: 1.0).opacity(0.12))
                .frame(width: 36, height: 36)
            Image(systemName: label == "Latest" ? "clock.fill" : "clock.arrow.circlepath")
                .font(.system(size: 15))
                .foregroundColor(Color(red: 0.3, green: 0.6, blue: 1.0))
        }
        VStack(alignment: .leading, spacing: 4) {
            Text(label)
                .font(.system(size: 15, weight: .semibold))
                .foregroundColor(.white)
            Text(entry.timestamp, formatter: TrendsView.timeFormatter)
                .font(.system(size: 12))
                .foregroundColor(.white.opacity(0.4))
            // Pressure mini-grid
            HStack(spacing: 8) {
                ForEach([
                    ("P0", entry.pressure0),
                    ("P1", entry.pressure1),
                    ("P2", entry.pressure2),
                    ("P3", entry.pressure3)
                ], id: \.0) { lbl, val in
                    VStack(spacing: 1) {
                        Text(lbl)
                            .font(.system(size: 9, weight: .semibold))
                            .foregroundColor(.white.opacity(0.3))
                        Text(String(format: "%.1f", val))
                            .font(.system(size: 11, weight: .bold, design: .monospaced))
                    }
                }
            }
        }
    }
}

```

```

                .foregroundColor(pressureStatus(for: val, sensor:
["P0", "P1", "P2", "P3"].firstIndex(of: lbl) ?? 0).color)
            }
        }
    }
    .padding(.top, 2)
    // IMU mini-row
    HStack(spacing: 8) {
        ForEach([
            ("Ax", entry.imuX),
            ("Ay", entry.imuY),
            ("Az", entry.imuZ)
        ], id: \.0) { lbl, val in
            Text("\(lbl): \(String(format: "%.2f", val))")
                .font(.system(size: 10, design: .monospaced))
                .foregroundColor(.white.opacity(0.3))
        }
    }
    Spacer()
}
.padding(.horizontal, 16)
.padding(.vertical, 14)
}
@ViewBuilder
private func divider() -> some View {
    Rectangle()
        .fill(Color.white.opacity(0.06))
        .frame(height: 1)
        .padding(.leading, 60)
}
}
struct TrendsView_Previews: PreviewProvider {
    static var previews: some View {
        TrendsView()
            .environmentObject(BluetoothManager())
    }
}

```

Section X. HomeView.swift

```
//  
// HomeView.swift  
// LymphSense  
//  
import SwiftUI  
import CoreBluetooth  
// MARK: - Data Model  
struct PressureData: Identifiable {  
    let id = UUID()  
    let x: Double  
    let y: Double  
    let z: Double  
    let pressure: Double  
}  
// MARK: - Data Manager  
@Observable  
final class PressureDataManager {  
    var dataPoints: [PressureData] = []  
    private var timer: Timer?  
    let goalPressure: Double = 50  
}  
// MARK: - Per-sensor pressure specs  
struct SensorSpec {  
    let low: Double  
    let high: Double  
    let tolerance: Double  
}  
private let sensorSpecs: [Int: SensorSpec] = [  
    0: SensorSpec(low: 18, high: 28, tolerance: 2), // Upper Arm    mmHg  
    1: SensorSpec(low: 35, high: 45, tolerance: 2), // Upper Middle mmHg  
    2: SensorSpec(low: 36, high: 46, tolerance: 2), // Lower Middle mmHg  
    3: SensorSpec(low: 40, high: 50, tolerance: 2), // Lower Arm    mmHg  
]  
func pressureStatus(for value: Double, sensor: Int) -> PressureStatus {  
    guard let spec = sensorSpecs[sensor] else { return .critical }  
    if value >= spec.low && value <= spec.high { return .good }  
    if value < spec.low {  
        return (spec.low - value) <= spec.tolerance ? .warning : .critical  
    } else {
```

```

        return (value - spec.high) <= spec.tolerance ? .warning : .critical
    }
}
// MARK: - Pressure Status
enum PressureStatus {
    case good, warning, critical
    var color: Color {
        switch self {
            case .good:    return Color(red: 0.035, green: 0.612, blue: 0.063)
            case .warning: return Color(red: 0.906, green: 0.906, blue: 0.078)
            case .critical: return Color(red: 0.839, green: 0.035, blue: 0.035)
        }
    }
    var label: String {
        switch self {
            case .good:    return "Optimal"
            case .warning: return "Elevated"
            case .critical: return "Critical"
        }
    }
}
// MARK: - Pressure Card
struct PressureCard: View {
    let label: String
    let sublabel: String
    let value: Double
    let sensor: Int
    let index: Int
    private var status: PressureStatus {
        pressureStatus(for: value, sensor: sensor)
    }
    private var barFill: Double {
        guard let spec = sensorSpecs[sensor], spec.high + spec.tolerance > 0 else { return 0 }
        let rangeMax = spec.high + spec.tolerance
        return min(max(value / rangeMax, 0), 1.0)
    }
    var body: some View {
        HStack(spacing: 16) {
            ZStack {
                Circle()

```

```

        .fill(status.color.opacity(0.15))
        .frame(width: 36, height: 36)
    Text("\index")
        .font(.system(size: 14, weight: .bold, design: .monospaced))
        .foregroundColor(status.color)
    }
    VStack(alignment: .leading, spacing: 2) {
        Text(label)
            .font(.system(size: 15, weight: .semibold))
            .foregroundColor(.white)
        Text(sublabel)
            .font(.system(size: 12, weight: .regular))
            .foregroundColor(.white.opacity(0.4))
    }
    Spacer()
    VStack(alignment: .trailing, spacing: 4) {
        Text(String(format: "%.1f mmHg", value))
            .font(.system(size: 16, weight: .bold, design: .monospaced))
            .foregroundColor(status.color)
        GeometryReader { geo in
            ZStack(alignment: .leading) {
                RoundedRectangle(cornerRadius: 2)
                    .fill(Color.white.opacity(0.08))
                RoundedRectangle(cornerRadius: 2)
                    .fill(status.color)
                    .frame(width: max(0, geo.size.width * barFill))
                    .animation(.spring(duration: 0.4), value: value)
            }
        }
        .frame(width: 80, height: 4)
        Text(status.label)
            .font(.system(size: 10, weight: .semibold))
            .foregroundColor(status.color)
            .tracking(0.8)
            .textCase(.uppercase)
    }
}
}
.padding(.horizontal, 16)
.padding(.vertical, 20)
.background(

```

```

        RoundedRectangle(cornerRadius: 14)
            .fill(Color.white.opacity(0.05))
            .overlay(
                RoundedRectangle(cornerRadius: 14)
                    .strokeBorder(status.color.opacity(0.2), lineWidth: 1)
            )
        )
    }
}

// MARK: - Summary Badge
struct SummaryBadge: View {
    let good: Int
    let warning: Int
    let critical: Int
    var body: some View {
        HStack(spacing: 12) {
            statPill(count: good, color: Color(red: 0.035, green: 0.612, blue: 0.063), label:
"Optimal")
            statPill(count: warning, color: Color(red: 0.906, green: 0.906, blue: 0.078), label:
"Elevated")
            statPill(count: critical, color: Color(red: 0.839, green: 0.035, blue: 0.035), label:
"Critical")
        }
    }
}

@ViewBuilder
func statPill(count: Int, color: Color, label: String) -> some View {
    HStack(spacing: 6) {
        Circle().fill(color).frame(width: 7, height: 7)
        Text("\(count) \(label)")
            .font(.system(size: 12, weight: .medium))
            .foregroundColor(.white.opacity(0.7))
    }
    .padding(.horizontal, 10)
    .padding(.vertical, 6)
    .background(color.opacity(0.1))
    .clipShape(Capsule())
}
}

// MARK: - Home View
struct HomeView: View {

```

```

@State private var manager = PressureDataManager()
@EnvironmentObject var bluetoothManager: BluetoothManager
@State private var pulse = false
private let sensorOrder: [Int] = [0, 1, 2, 3]
private func latestValue(for sensor: Int) -> Double? {
    guard let latest = bluetoothManager.receivedDataHistory.last else { return nil }
    switch sensor {
    case 0: return latest.pressure0
    case 1: return latest.pressure1
    case 2: return latest.pressure2
    case 3: return latest.pressure3
    default: return nil
    }
}
private func sensorLabel(_ sensor: Int) -> String {
    switch sensor {
    case 0: return "Upper Arm"
    case 1: return "Upper Middle Arm"
    case 2: return "Lower Middle Arm"
    case 3: return "Lower Arm"
    default: return "Unknown"
    }
}
private func sensorSublabel(_ sensor: Int) -> String {
    switch sensor {
    case 0: return "Proximal — Zone A"
    case 1: return "Proximal — Zone B"
    case 2: return "Distal — Zone C"
    case 3: return "Distal — Zone D"
    default: return ""
    }
}
var body: some View {
    ZStack {
        Color(red: 0.07, green: 0.07, blue: 0.1).ignoresSafeArea()
        RadialGradient(
            colors: [Color(red: 0.15, green: 0.25, blue: 0.45).opacity(0.4), .clear],
            center: .top, startRadius: 0, endRadius: 400
        )
        .ignoresSafeArea()
    }
}

```



```

ScrollView(showsIndicators: false) {
  VStack(spacing: 0) {
    // MARK: Header
    HStack {
      VStack(alignment: .leading, spacing: 4) {
        Text("LymphSense")
          .font(.system(size: 28, weight: .bold, design: .rounded))
          .foregroundColor(.white)
        Text("Pressure Monitoring")
          .font(.system(size: 13, weight: .regular))
          .foregroundColor(.white.opacity(0.4))
          .tracking(1.5)
          .textCase(.uppercase)
      }
      Spacer()
    }
    HStack(spacing: 6) {
      ZStack {
        if bluetoothManager.isConnected {
          Circle()
            .fill(Color(red: 0.035, green: 0.612, blue: 0.063))
            .frame(width: 8, height: 8)
            .scaleEffect(pulse ? 1.4 : 1.0)
            .opacity(pulse ? 0.3 : 1.0)
            .animation(.easeInOut(duration: 1.2).repeatForever(), value: pulse)
        }
        Circle()
          .fill(bluetoothManager.isConnected
            ? Color(red: 0.035, green: 0.612, blue: 0.063)
            : Color(red: 0.839, green: 0.035, blue: 0.035))
          .frame(width: 8, height: 8)
      }
      Text(bluetoothManager.isConnected ? "Connected" : "Disconnected")
        .font(.system(size: 12, weight: .semibold))
        .foregroundColor(.white.opacity(0.85))
    }
  }
  .padding(.horizontal, 12)
  .padding(.vertical, 8)
  .background(Color.white.opacity(0.07))
  .clipShape(Capsule())
  .overlay(Capsule().strokeBorder(Color.white.opacity(0.1), lineWidth: 1))
}

```

```

}
.padding(.horizontal, 24)
.padding(.top, 20)
.padding(.bottom, 20)
// MARK: Arm Visualization
ArmPressureView(
  values: sensorOrder.map { latestValue(for: $0) },
  isConnected: bluetoothManager.isConnected
)
.frame(width: 220, height: 160)
.padding(.vertical, 4)
.padding(.horizontal, 12)
.padding(.bottom, 12)
Rectangle()
  .fill(Color.white.opacity(0.06))
  .frame(height: 1)
  .padding(.horizontal, 24)
  .padding(.bottom, 10)
// MARK: Sensor Cards
VStack(spacing: 6) {
  HStack {
    Text("Pressure Regions")
      .font(.system(size: 11, weight: .semibold))
      .foregroundColor(.white.opacity(0.35))
      .tracking(1.5)
      .textCase(.uppercase)
    Spacer()
    Text("mmHg")
      .font(.system(size: 11, weight: .medium))
      .foregroundColor(.white.opacity(0.2))
  }
  .padding(.horizontal, 24)
  .padding(.bottom, 4)
  ForEach(Array(sensorOrder.enumerated()), id: \.element) { i, sensor in
    let value = latestValue(for: sensor) ?? 0.0
    PressureCard(
      label: sensorLabel(sensor),
      sublabel: sensorSublabel(sensor),
      value: value,
      sensor: sensor,

```

```

        index: i + 1
    )
    .padding(.horizontal, 16)
    .transition(.move(edge: .bottom).combined(with: .opacity))
}
}
Text("Updates in real time")
    .font(.system(size: 11))
    .foregroundColor(.white.opacity(0.2))
    .padding(.top, 20)
    .padding(.bottom, 40)
}
}
}
    .onAppear { pulse = true }
}
}
#Preview {
    HomeView()
        .environmentObject(BluetoothManager())
}

```

Section XI. MainTabView.swift

```
//  
// MainTabView.swift  
// LymphSense  
//  
import SwiftUI  
struct MainTabView: View {  
    // Access the shared manager from the environment  
    @EnvironmentObject var bluetoothManager: BluetoothManager  
    var body: some View {  
        TabView {  
            HomeView()  
                .tabItem {  
                    Label("Home", systemImage: "house")  
                }  
  
            DataView()  
                .tabItem {  
                    Label("Data", systemImage: "gauge.chart.lefthalf.righthalf")  
                }  
  
            TrendsView()  
                .tabItem {  
                    Label("Trends", systemImage: "waveform.path.ecg.rectangle")  
                }  
  
            ProfileView()  
                .tabItem {  
                    Label("Profile", systemImage: "person")  
                }  
        }  
    }  
}  
// -----  
// Preview  
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}
```

Section XII. ProfileView.swift

```
//
// ProfileView.swift
// LymphSense
//
import SwiftUI
struct ProfileView: View {
    @EnvironmentObject var bluetoothManager: BluetoothManager
    @State private var isShowingClearAlert = false
    @State private var pulse = false
    var body: some View {
        ZStack {
            // Background
            Color(red: 0.07, green: 0.07, blue: 0.1)
                .ignoresSafeArea()
            // Subtle radial glow
            RadialGradient(
                colors: [Color(red: 0.15, green: 0.25, blue: 0.45).opacity(0.4), .clear],
                center: .top,
                startRadius: 0,
                endRadius: 400
            )
            .ignoresSafeArea()
            ScrollView(showsIndicators: false) {
                VStack(spacing: 0) {
                    // MARK: Header
                    HStack {
                        VStack(alignment: .leading, spacing: 4) {
                            Text("Profile")
                                .font(.system(size: 28, weight: .bold, design: .rounded))
                                .foregroundColor(.white)
                            Text("Device & Settings")
                                .font(.system(size: 13, weight: .regular))
                                .foregroundColor(.white.opacity(0.4))
                                .tracking(1.5)
                                .textCase(.uppercase)
                        }
                        Spacer()
                        // Connection pill
                        HStack(spacing: 6) {
```

```

ZStack {
  if bluetoothManager.isConnected {
    Circle()
      .fill(Color(red: 0.035, green: 0.612, blue: 0.063))
      .frame(width: 8, height: 8)
      .scaleEffect(pulse ? 1.4 : 1.0)
      .opacity(pulse ? 0.3 : 1.0)
      .animation(.easeInOut(duration: 1.2).repeatForever(), value: pulse)
  }
  Circle()
    .fill(bluetoothManager.isConnected
      ? Color(red: 0.035, green: 0.612, blue: 0.063)
      : Color(red: 0.839, green: 0.035, blue: 0.035))
    .frame(width: 8, height: 8)
  }
  Text(bluetoothManager.isConnected ? "Connected" : "Disconnected")
    .font(.system(size: 12, weight: .semibold))
    .foregroundColor(.white.opacity(0.85))
  }
  .padding(.horizontal, 12)
  .padding(.vertical, 8)
  .background(Color.white.opacity(0.07))
  .clipShape(Capsule())
  .overlay(Capsule().strokeBorder(Color.white.opacity(0.1), lineWidth: 1))
  }
  .padding(.horizontal, 24)
  .padding(.top, 20)
  .padding(.bottom, 28)
  // MARK: Section — Bluetooth Device
  sectionHeader("Bluetooth Device")
  VStack(spacing: 1) {
    // Connect row
    NavigationLink(destination: BluetoothView(bluetoothManager:
bluetoothManager)) {
      profileRow(
        icon: "dot.radiowaves.left.and.right",
        iconColor: Color(red: 0.3, green: 0.6, blue: 1.0),
        title: "Connect to Device",
        subtitle: "Scan & pair your sensor band",
        showChevron: true

```

```

        )
    }
    divider()
    // Status row
    profileRow(
        icon: bluetoothManager.isConnected ? "checkmark.circle.fill" :
"xmark.circle.fill",
        iconColor: bluetoothManager.isConnected
            ? Color(red: 0.035, green: 0.612, blue: 0.063)
            : Color(red: 0.839, green: 0.035, blue: 0.035),
        title: "Connection Status",
        subtitle: bluetoothManager.isConnected ? "Device is connected" : "No device
found",

        showChevron: false
    )
}
.background(
    RoundedRectangle(cornerRadius: 16)
        .fill(Color.white.opacity(0.05))
        .overlay(
            RoundedRectangle(cornerRadius: 16)
                .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
        )
)
.padding(.horizontal, 16)
.padding(.bottom, 20)
// MARK: Section — Notifications
sectionHeader("Notifications")
VStack(spacing: 1) {
    Button {
        NotificationManager.shared.scheduleLocalNotification()
    } label: {
        profileRow(
            icon: "bell.badge.fill",
            iconColor: Color(red: 1.0, green: 0.6, blue: 0.1),
            title: "Send Test Notification",
            subtitle: "Trigger a local push notification",
            showChevron: false
        )
    }
}

```

```

    }
    .background(
      RoundedRectangle(cornerRadius: 16)
        .fill(Color.white.opacity(0.05))
        .overlay(
          RoundedRectangle(cornerRadius: 16)
            .strokeBorder(Color.white.opacity(0.08), lineWidth: 1)
        )
    )
    .padding(.horizontal, 16)
    .padding(.bottom, 20)
    // MARK: Section — Data
    sectionHeader("Data Management")
    VStack(spacing: 1) {
      Button(role: .destructive) {
        isShowingClearAlert = true
      } label: {
        profileRow(
          icon: "trash.fill",
          iconColor: Color(red: 0.839, green: 0.035, blue: 0.035),
          title: "Clear All Data History",
          subtitle: "Permanently delete recorded Bluetooth data",
          showChevron: false
        )
      }
    }
  }
  .background(
    RoundedRectangle(cornerRadius: 16)
      .fill(Color.white.opacity(0.05))
      .overlay(
        RoundedRectangle(cornerRadius: 16)
          .strokeBorder(Color(red: 0.839, green: 0.035, blue: 0.035).opacity(0.2),
lineWidth: 1)
      )
  )
  .padding(.horizontal, 16)
  .padding(.bottom, 40)
}
}
}

```



```

        .onAppear { pulse = true }
        .alert("Confirm Data Deletion", isPresented: $isShowingClearAlert) {
            Button("Delete", role: .destructive) {
                bluetoothManager.clearHistory()
            }
            Button("Cancel", role: .cancel) { }
        } message: {
            Text("Are you sure you want to permanently delete ALL recorded Bluetooth data? This
cannot be undone.")
        }
    }
}
// MARK: - Reusable subviews
@ViewBuilder
private func sectionHeader(_ title: String) -> some View {
    HStack {
        Text(title)
            .font(.system(size: 11, weight: .semibold))
            .foregroundColor(.white.opacity(0.35))
            .tracking(1.5)
            .textCase(.uppercase)
        Spacer()
    }
    .padding(.horizontal, 28)
    .padding(.bottom, 8)
}
@ViewBuilder
private func divider() -> some View {
    Rectangle()
        .fill(Color.white.opacity(0.06))
        .frame(height: 1)
        .padding(.leading, 60)
}
@ViewBuilder
private func profileRow(
    icon: String,
    iconColor: Color,
    title: String,
    subtitle: String,
    showChevron: Bool
) -> some View {

```

```

HStack(spacing: 14) {
  // Icon badge
  ZStack {
    RoundedRectangle(cornerRadius: 8)
      .fill(iconColor.opacity(0.15))
      .frame(width: 36, height: 36)
    Image(systemName: icon)
      .font(.system(size: 16, weight: .medium))
      .foregroundColor(iconColor)
  }
  VStack(alignment: .leading, spacing: 2) {
    Text(title)
      .font(.system(size: 15, weight: .semibold))
      .foregroundColor(.white)
    Text(subtitle)
      .font(.system(size: 12))
      .foregroundColor(.white.opacity(0.4))
  }
  Spacer()
  if showChevron {
    Image(systemName: "chevron.right")
      .font(.system(size: 12, weight: .semibold))
      .foregroundColor(.white.opacity(0.25))
  }
}
.padding(.horizontal, 16)
.padding(.vertical, 14)
}
}
struct ProfileView_Previews: PreviewProvider {
  static var previews: some View {
    ProfileView()
      .environmentObject(BluetoothManager())
  }
}

```

Section XIII. LymphSenseApp.swift

```
//  
// LymphSenseApp.swift  
// LymphSense  
//  
import SwiftUI  
@main  
struct LymphSenseApp: App {  
    init() {  
        // Must request permission here!  
        NotificationManager.shared.requestAuthorization()  
    }  
  
    var body: some Scene {  
        WindowGroup {  
            ContentView()  
        }  
    }  
}
```

Section XIV. Calibration.ino

```
#include <Arduino.h>
#include <Wire.h>

// ----- TCA9548A -----
#define TCAADDR 0x70 // default TCA9548A address (A0/A1/A2 = GND)

void tcaSelect(uint8_t i) {
  if (i > 7) return;
  Wire.beginTransmission(TCAADDR);
  Wire.write(1 << i);
  Wire.endTransmission();
}

// Optional: disable all channels (sometimes helps when debugging)
static void tcaDisableAll() {
  Wire.beginTransmission(TCAADDR);
  Wire.write(0x00);
  Wire.endTransmission();
}

// ----- BLUETOOTH -----
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

BLECharacteristic *txCharacteristic;
bool deviceConnected = false;

// Nordic UART UUIDs (iPhone compatible)
#define SERVICE_UUID "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"

class MyServerCallbacks: public BLEServerCallbacks {
  void onConnect(BLEServer* pServer) { deviceConnected = true; }
  void onDisconnect(BLEServer* pServer) { deviceConnected = false; }
};

// ----- SENSOR -----
```

```

const uint8_t SENSOR_ADDR = 0x28;           // 40 dec
const uint8_t SENSOR_CMD[3] = {0xAA, 0x00, 0x00};
uint8_t data[7];

char bleBuffer[50];

const double outputmax = 15099494;
const double outputmin = 1677722;
const double pmax = 1; // set to your sensor range max
const double pmin = 0; // set to your sensor range min

char printBuffer[220], cBuff[20], percBuff[20], pBuff[20], tBuff[20];

// Channels you are using:
const uint8_t channels[] = {4, 5, 6, 7}; // channel 6 was not working, use channel 0 instead!
const uint8_t NUM_CH = sizeof(channels) / sizeof(channels[0]);

static bool sensorPresentOnChannel(uint8_t ch) {
    tcaSelect(ch);
    Wire.beginTransmission(SENSOR_ADDR);
    uint8_t err = Wire.endTransmission(); // 0 = ACK
    return (err == 0);
}

static bool readSensorOnChannel(uint8_t ch, uint8_t outData[7]) {
    tcaSelect(ch);

    // Send command
    Wire.beginTransmission(SENSOR_ADDR);
    Wire.write(SENSOR_CMD, sizeof(SENSOR_CMD));
    uint8_t err = Wire.endTransmission(); // 0 = success
    if (err != 0) return false;

    delay(10);

    // Request 7 bytes
    tcaSelect(ch); // re-select before request (good practice)
    uint8_t got = Wire.requestFrom((int)SENSOR_ADDR, 7);
    if (got != 7) return false;

```

```

for (uint8_t i = 0; i < 7; i++) {
    if (!Wire.available()) return false;
    outData[i] = Wire.read();
}
return true;
}

// ----- Calibration -----
// x = raw pressure from sensor
// returns calibrated pressure
double calibratePressure(uint8_t sensorIndex, double x) {
    switch (sensorIndex) {
        case 0: // Sensor 1 / channel 0
            return 85.0316*x + 2.73572;

        case 1: // Sensor 2 / channel 1
            return 82.7685*x + 5.62809;

        case 2: // Sensor 3 / channel 2
            return 77.5495*x + 1.32534;

        case 3: // Sensor 4 / channel 3
            return 108.41*x - 1.80684;

        default:
            return x; // fallback: no calibration
    }
}

// ----- ADXL343 IMU -----
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL343.h>

#define ADXL343_SCK 13
#define ADXL343_MISO 12
#define ADXL343_MOSI 11
#define ADXL343_CS 10

/* Assign a unique ID to this sensor at the same time */
/* Uncomment following line for default Wire bus */

```

```

Adafruit_ADXL343 accel = Adafruit_ADXL343(12345);

/* NeoTrellis M4, etc.          */
/* Uncomment following line for Wire1 bus */
//Adafruit_ADXL343 accel = Adafruit_ADXL343(12345, &Wire1);

/* Uncomment for software SPI */
//Adafruit_ADXL343 accel = Adafruit_ADXL343(ADXL343_SCK, ADXL343_MISO,
ADXL343_MOSI, ADXL343_CS, 12345);

/* Uncomment for hardware SPI */
//Adafruit_ADXL343 accel = Adafruit_ADXL343(ADXL343_CS, &SPI, 12345);

void displayDataRate(void)
{
  Serial.print ("Data Rate:  ");

  switch(accel.getDataRate())
  {
    case ADXL343_DATARATE_3200_HZ:
      Serial.print ("3200 ");
      break;
    case ADXL343_DATARATE_1600_HZ:
      Serial.print ("1600 ");
      break;
    case ADXL343_DATARATE_800_HZ:
      Serial.print ("800 ");
      break;
    case ADXL343_DATARATE_400_HZ:
      Serial.print ("400 ");
      break;
    case ADXL343_DATARATE_200_HZ:
      Serial.print ("200 ");
      break;
    case ADXL343_DATARATE_100_HZ:
      Serial.print ("100 ");
      break;
    case ADXL343_DATARATE_50_HZ:
      Serial.print ("50 ");
      break;
  }
}

```

```

    case ADXL343_DATARATE_25_HZ:
        Serial.print ("25 ");
        break;
    case ADXL343_DATARATE_12_5_HZ:
        Serial.print ("12.5 ");
        break;
    case ADXL343_DATARATE_6_25HZ:
        Serial.print ("6.25 ");
        break;
    case ADXL343_DATARATE_3_13_HZ:
        Serial.print ("3.13 ");
        break;
    case ADXL343_DATARATE_1_56_HZ:
        Serial.print ("1.56 ");
        break;
    case ADXL343_DATARATE_0_78_HZ:
        Serial.print ("0.78 ");
        break;
    case ADXL343_DATARATE_0_39_HZ:
        Serial.print ("0.39 ");
        break;
    case ADXL343_DATARATE_0_20_HZ:
        Serial.print ("0.20 ");
        break;
    case ADXL343_DATARATE_0_10_HZ:
        Serial.print ("0.10 ");
        break;
    default:
        Serial.print ("???? ");
        break;
}
Serial.println(" Hz");
}

void displayRange(void)
{
    Serial.print ("Range:      +/- ");

    switch(accel.getRange())
    {

```



```

    case ADXL343_RANGE_16_G:
        Serial.print ("16 ");
        break;
    case ADXL343_RANGE_8_G:
        Serial.print ("8 ");
        break;
    case ADXL343_RANGE_4_G:
        Serial.print ("4 ");
        break;
    case ADXL343_RANGE_2_G:
        Serial.print ("2 ");
        break;
    default:
        Serial.print ("?? ");
        break;
}
Serial.println(" g");
}

// ----- SETUP -----

void setup() {
    Serial.begin(115200);
    while (!Serial);

    Wire.begin();
    delay(1000);

    Serial.println("Accelerometer Test"); Serial.println("");

    /* Initialize the IMU */
    tcaSelect(7);
    delay(10);

    if(!accel.begin())
    {
        /* There was a problem detecting the ADXL343 ... check your connections */
        Serial.println("Ooops, no ADXL343 detected ... Check your wiring!");
        while(1);
    }
}

```

```

Serial.println(F("\nTCA9548A + 4x sensor reader (channels 4-7)"));
Serial.println(F("CH\tStatus\tRaw24(P)\t\tCounts\t%FS\tPressure\tTempC"));

// Quick presence report
for (uint8_t i = 0; i < NUM_CH; i++) {
    uint8_t ch = channels[i];
    Serial.print(F("Channel "));
    Serial.print(ch);
    Serial.print(F(": "));
    Serial.println(sensorPresentOnChannel(ch) ? F("FOUND 0x28") : F("NOT FOUND"));
}

/* Set the range to whatever is appropriate for your project */
accel.setRange(ADXL343_RANGE_16_G);
// accel.setRange(ADXL343_RANGE_8_G);
// accel.setRange(ADXL343_RANGE_4_G);
// accel.setRange(ADXL343_RANGE_2_G);

/* Display some basic information on this sensor */
accel.printSensorDetails();
displayDataRate();
displayRange();
Serial.println("");

Serial.println(F("\nTCA9548A + 4x sensor reader (channels 4-7)"));
Serial.println(F("CH\tStatus\tRaw24(P)\t\tCounts\t%FS\tPressure\tTempC"));

/* Bluetooth */
// BLE Setup
BLEDevice::init("ESP32-PressureArray");
BLEServer *pServer = BLEDevice::createServer();
pServer->setCallbacks(new MyServerCallbacks());

BLEService *pService = pServer->createService(SERVICE_UUID);

txCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID_TX,
    BLECharacteristic::PROPERTY_NOTIFY
);

```

```

txCharacteristic->addDescriptor(new BLE2902());

pService->start();
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->start();

tcaDisableAll();
}

// ----- LOOP -----

void loop() {
    static unsigned long lastMillis = 0;
    unsigned long now = millis();

    if (now - lastMillis < 1000) return;
    lastMillis = now;

    float timeSeconds = now / 1000.0;

    String values[5]; // array with pressure and imu values

    for (uint8_t ch = 0; ch < 8; ch++) {
        if (ch == 4 || ch == 5 || ch == 6) {
            continue;
        }

        tcaSelect(ch);
        //Serial.print("\nChannel ");
        //Serial.println(ch);

        if (ch < 4) {
            if (!readSensorOnChannel(ch, data)) {
                values[ch] = String(-1.0);
                continue;
            }
        }

        // Raw pressure counts from sensor bytes

```

```

double press_counts = (double)(((uint32_t)data[1] << 16) |
                                ((uint32_t)data[2] << 8) |
                                (uint32_t)data[3]);

// Convert counts to raw pressure value
double rawPressure = ((press_counts - outputmin) * (pmax - pmin)) /
                    (outputmax - outputmin) + pmin;

double pressure = calibratePressure(ch, rawPressure);

char temp[20];
snprintf(temp, sizeof(temp), "%.3f", pressure);
values[ch] = String(temp);

delay(50); // small spacing between channels
} else if (ch == 7) {
    sensors_event_t event;
    accel.getEvent(&event);

    char temp[80];
    snprintf(temp, sizeof(temp), "%.3f, %.3f, %.3f",
             event.acceleration.x,
             event.acceleration.y,
             event.acceleration.z);
    values[4] = String(temp);
}
}

// Build the full CSV string
char blePacket[128];
snprintf(blePacket, sizeof(blePacket), "%s, %s, %s, %s, %s, %.2f",
        values[0].c_str(),
        values[1].c_str(),
        values[2].c_str(),
        values[3].c_str(),
        values[4].c_str(),
        timeSeconds);

// Print to Serial for debugging
Serial.println(blePacket);

```

```
// Send over BLE — this is what was missing
if (deviceConnected) {
    txCharacteristic->setValue(blePacket);
    txCharacteristic->notify();
}

tcaDisableAll();
//delay(200);
delay(1000);
}
```