2HADRAD

Portable Life Detection Radar

[Hitt, **H**ankinson, **A**ndersen, **D**'Souza **– Rad**ar**]**

Group 32 Authors:

Ryan Andersen	Bailey Hitt	Shannon Hankinson	Joshua D'Souza
Electrical Engineering	Electrical Engineering	Electrical Engineering	Computer Engineering

Review Committee:

Dr. Masoumeh Kalantari-Khandani	ECE	Lecturer
Dr. Enxia Zhang	ECE	Assistant Professor
Dr. Mark Maddox	ECE	Instructor
Dr. Mike Borowicz	ECE	Associate Professor

Advisor:

Professor Sreeram Sundaresh, Ph.D.

Department of Electrical and Computer Engineering, University of Central Florida

EEL4914: Senior Design I

Fall 2025

UCF | College of Engineering and Computer Science

UNIVERSITY OF CENTRAL FLORIDA

Table of Contents

Table of Contents ..... *i*

List of Tables ..... *viii*

List of Figures ..... *ix*

CHAPTER 1 ..... *1*

1.1. Executive Summary **1**

CHAPTER 2 ..... *2*

2.1. PROJECT BACKGROUND AND MOTIVATION **2**

2.2. PROJECT FUNCTION **3**

2.3. Current Commercial Technologies and Existing Projects **4**

2.3.1. Portable Sign of Life Identifier (Department of Homeland Security) 4

2.3.2. Integration of Multi Sensor Detection Technologies 4

2.3.3. Commercial Through Wall Radar Systems: Camero Xaver and Vayyar Imaging 5

2.3.4. Infrared Assisted Radar Concepts 5

2.3.5. Summary 6

2.4. PROJECT OBJECTIVES AND GOALS **6**

2.4.1. GOALS 6

2.4.2. OBJECTIVES 7

2.4.3. REQUIREMENT SPECIFICATIONS 8

2.5. RELATED STANDARDS **10**

2.6. PROJECT BLOCK DIAGRAM **10**

2.7. HOUSE OF QUALITY **15**

2.8. PROJECT BUDGET AND FINANCING **15**

2.9. INITIAL PROJECT MILESTONES **16**

2.10. PROTOTYPE **17**

CHAPTER 3 ..... *19*

3.1. Location/Tracking **19**

3.1.1. Inertial Measurement Unit 19

3.1.2. Inertial Navigation System 20

3.1.3. Ultra-Wideband (UWB) 21

3.1.4. IR and Radar Module 21

3.1.5. Barometric Pressure Sensor 22

3.1.6. Magnetometers 22

3.2. Location Part Selection 23
3.2.1. Inertial Navigation System 23
3.2.1.1. M5Stack M135 Module 23
3.2.1.2. STMicroelectronics Teseo-VIC3DA Module 23
3.2.1.3. U-blox NEO-M9V 24
3.2.1.4. U-blox ZED-F9R-03B 24
3.2.1.5. Custom INS Stack 25
3.2.1.6. INS Selection Justification 27
3.2.2. Barometer 28
3.2.2.1. Bosch BMP390 28
3.2.2.2. Infineon DPS310 28
3.2.2.3. Bosch BMP280 28
3.2.2.4. ST LPS22HB 29
3.2.2.5. Barometer Selection Justification 29
3.2.3. Magnetometer 30
3.2.3.1. MEMSIC MMC5983MA 30
3.2.3.2. ST LIS3MD 30
3.2.3.3. QST QMC5883L 30
3.2.3.4. Bosch BMM150 30
3.2.3.5. Magnetometer Selection Justification 31
3.3. Life Detection Modules 31
3.3.1. Ultra-Wideband (UWB) Modules 32
3.3.2. Low Frequency RF Modules 33
3.3.3. Seismic Sensors 33
3.3.4. Radar Module 34
3.3.5. Infrared Thermal Sensor 34
3.3.6. CO-2 Sensor 35
3.4. Life Detection Module Part Selection 35
3.4.1. Ultra-Wideband Modules 36
3.4.1.1. DWM3000EVB 36
3.4.1.2. ULM3-STM32 + DWM3000 36
3.4.1.3. DWM1000 36
3.4.1.4. DWM3000 36
3.4.1.5. UWB Selection Justification 37
3.4.2. Geophones 38
3.4.2.1. Geophone SM-24 38
3.4.2.2. SM4 Horizontal Geophone 38
3.4.2.3. EG-4.5-II Geophone 39
3.4.2.4. Geophone Selection Justification 39
3.4.3. Radar Modules 39
3.4.3.1. Infineon BGT60TR13C 39
3.4.3.2. Infineon BGT60ATR24C 40
3.4.3.3. TI IWR6843AOP EVM 41
3.4.3.4. TI AWR1843BOOST EVM 42

3.4.3.5. Waveshare HMMD 24 GHz mmWave Radar 42
3.4.3.6. FMCW Principle and Range-Doppler Processing 43
3.4.3.7. Micro-Doppler Extraction for Vital signs 44
3.4.3.8. On Chip Processing and Firmware Framework 45
3.4.3.9. Chirp Design, Data Handling, and Integration 45
3.4.3.10. Mechanical, EMI, and Thermal Design Constraints 46
3.4.3.11. Radar Selection Justification 47
3.4.4. IR Sensor 48
3.4.4.1. AMG8833 IR Thermal Camera (Grid EYE) 48
3.4.4.2. MLX90640 32×24 Thermal Camera Array 49
3.4.4.3. MLX90641 16×12 Thermal Array 49
3.4.4.4. FLIR Lepton 3.5 Thermal Imaging Module 49
3.4.4.5. Infrared Temperature Sensor (MLX90614) 50
3.4.4.6. Limitations 50
3.4.4.7. Selected IR Sensor Justification 51

3.5. Safety and RF Exposure Management 52

3.6. Safety and RF Exposure Management Part Selection 52
3.6.1. HC-SR04 Ultra Sonic Sensor (USS) 52
HC-SR501 Passive Infrared Sensor 52
3.6.2. VL53L0X Time of Flight (ToF) Laser Distance Sensor 53
3.6.3. Safety and RF Exposure Management Part Selection Justification 53

3.7. User Interface 54
3.7.1. Pushbutton 54
3.7.2. LED Indicator 55
3.7.3. Buzzer 56
3.7.4. Speaker 56
3.7.5. Display Screens 56

3.8. User Interface Parts Selection 57
3.8.1. Pushbutton 57
3.8.1.1. Apem 1ZW0913611806 57
3.8.1.2. E-Switch PV6F24011 57
3.8.1.3. C&K PTS645VL58-2 LFS 58
3.8.1.4. Pushbutton Selection Justification 58
3.8.2. LED 59
3.8.2.1. Kingbright KP-1608 Series (Green, Yellow, Red, Blue, Amber) 59
3.8.2.2. Lite-On LTST-C191 Series (Red, Orange, Yellow, Green, Blue) 59
3.8.2.3. OSRAM R976 Series (Red, Yellow, Orange, Blue) 59
3.8.2.4. LED Selection Justification 60
3.8.3. Buzzer 61
3.8.3.1. CUI CMT-1203-SMT-TR 61
3.8.3.2. Mallory Sonalert PS-1206P 61
3.8.3.3. Soberton WT-1205 61
3.8.3.4. SameSky CMI-1295-03TH 61

3.8.3.5. Buzzer Selection Justification 62
3.8.4. Speaker 62
3.8.4.1. MAX98357A 62
3.8.4.2. PAM8302 63
3.8.4.3. LM386 63
3.8.4.4. Speaker Selection Justification 63
3.8.5. Display Screens 64
3.8.5.1. ILI9341 TFT LCD 64
3.8.5.2. 4.3'' 800x490 RA8875 TFT LCD Module 64
3.8.5.3. 4.3” RA8875 Capacitive Touch Module 65
3.8.5.4. 4.3” 800x490 IPS TFT LCD 65
3.8.5.5. 4.3” 740-nit Sunlight Readable TFT 65
3.8.5.6. 4.3” Capacitive Touch LCD 66
3.8.5.7. LCD Selection Justification 66

3.9. Processing and Control Modules 67
3.9.1. Microcontroller Unit 67
3.9.2. Microprocessor 67
3.9.3. System-on-Chip 68
3.9.4. MCU and Co-Processor Architecture 68
3.9.5. Field-Programmable Gate Array 68
3.9.6. Single-Board Computer 69

3.10. Processing and Control Module Part Selection 69
3.10.1. ESP 32 69
3.10.2. STM32 70
3.10.3. RasberryPi 70
3.10.4. MCU Selection Justification 71

3.11. MCU Part Comparison 72
3.11.1. ESP32-WROOM-32 72
3.11.2. ESP32-S3-WROOM-1 73
3.11.3. ESP32-C6-WROOM-1 74
3.11.4. MCU Final Selection Justification 75

3.12. Power Supply 76
3.12.1. MCU7 Baseboard 76
3.12.2. LCD Display 76
3.12.3. ESP32-WROOM 77
3.12.4. Inertial Navigation System 77
3.12.5. Ultra-Wide Band 78
3.12.6. Infrared Sensor 78
3.12.7. Low Power Supporting Components 78

3.13. Power Supply Part Selection 79
3.13.1. Battery Types 79
3.13.1.1. Lithium-Ion (Li-ion) 79
3.13.1.2. Lithium-Polymer (LiPo) 79

3.13.1.3. Nickel-Cadmium (NiCd) 80
3.13.1.4. Nickel-Metal Hydride (NiMH) 80
3.13.1.5. Sealed Lead-Acid (SLA) 80
3.13.2. Battery Selection and Justification 80
3.13.3. Voltage Regulation 81
3.13.3.1. LM34936 82
3.13.3.2. S7V8F3 82
3.13.3.3. LM33620 82
3.13.3.4. LM3478 82
3.13.3.5. LM2576 83
3.13.3.6. LM2731 83
3.13.3.7. Regulator Selection and Justification 84
3.13.3.8. Additional Required Power Hardware 85
3.14. Communication Protocols 85
3.14.1. Radar Module (TI IWR6843AOP) 86
3.14.2. UWB Signal Processing (DWM3000) 87
3.14.3. Infrared Thermal Sensor (MLX90640) 88
3.14.4. Inertial Navigation System 89
3.14.5. User Interface and Feedback 90
3.14.6. Data Fusion and Algorithm 91
3.14.7. Communication Summary 93
3.15. Communication Frameworks for Smartphone App Development 94
3.15.1. Bluetooth Low Energy (BLE) 94
3.15.2. WiFi (HTTP / WebSocket) 94
3.15.3. MQTT (Message Queuing Telemetry Transport) 95
3.15.4. USB (Wired Communication) 95
3.16. Smartphone Application Development 96
3.16.1. Development Infrastructure Comparison 96
3.16.2. Android Technology Comparison 96
3.16.2.1. Android Native (Kotlin + Jetpack Compose) 96
3.16.2.2. Android Native (Java + XML Layouts) 97
3.16.2.3. Android + MIT App Inventor / Kodular (Visual Builder) 97
3.16.2.4. Android with Flutter (Dart) 97
3.16.2.5. Android Native (Java + XML Layouts) 98
3.16.2.6. Android + MIT App Inventor / Kodular (Visual Builder) 98
3.16.2.7. Android with Flutter (Dart) 98
CHAPTER 4 99
4.1. Introduction to Standards and Design Constraints 99
4.1.1. FCC Part 15 – Radio Frequency Devices 100
4.1.2. IEEE 802.15.4 UWB Standard (802.15.4a / 802.15.4z) 101
4.1.3. The 2HADRAD system GNSS System Standards (GPS compliance) 102
4.1.4. I2C / SPI Interface Standards (Serial Bus Communication) 103
4.1.5. IEC Safety and EMC Standards (ESD, EMI mitigation) 103

4.1.6. ISO / IEC Quality and Testing Standards 104
4.1.7. Power Constraints 104
4.1.8. Thermal Constraints 106
4.1.9. Mechanical / Size Constraints 107
4.1.10. Cost Constraints 108
4.1.11. Environmental Constraints 108
4.1.12. Computational Constraints 109
4.1.13. Ethical / Safety Constraints 109

CHAPTER 5 109

5.1. Overview 109

5.2. Advantages and Limitations 110
5.2.1. Advantages 110
5.2.2. Limitations 110

5.3. Case Studies 111
5.3.1. Case Study 1: Power Management Optimization 111
5.3.2. Case Study 2: Signal Filtering in Noisy Environments 111
5.3.3. Case Study 3: Communication Protocol Selection 112
5.3.4. Case Study 4: Enclosure Material Selection 112
5.3.5. Case Study 5: Data Visualization on Embedded Displays 113

CHAPTER 6 114

6.1. Hardware Block Diagram 114

6.2. Sensor Subsystem 116
6.2.1. Radar Module 116
6.2.2. INS Module 116
6.2.3. IR Sensor 117
6.2.4. UWB Sensor 118
6.2.5. USS 120

6.3. MCU Configuration 120
6.3.1. USB to UART 123

6.4. User Interface 124

6.5. Power Subsystem 125

CHAPTER 7 128

7.1. ESP32 Software Design 129
7.1.1. User Interface 130
7.1.2. Thermal Map Operating Mode 132
7.1.3. Thermal Classification Pipeline 133
7.1.4. Radar Operating Mode 135
7.1.5. INS Operating Mode 137
7.1.6. Scan Operating Mode 139

7.1.7. Smartphone App Software Design 141
7.1.8. iOS Companion Application 142
7.1.9. Anchor Software Unit 143

CHAPTER 8 ..... *145*

8.1. PCB **145**

8.2. Prototype Construction **150**

8.3. Components **153**

CHAPTER 9 ..... *154*

9.1. Hardware Testing **154**
9.1.1. MCU 154
9.1.2. Radar Module 154
9.1.3. Inertial Navigation System 155
9.1.4. Infrared Thermal Sensor 155
9.1.5. Ultrasonic Sensor 156
9.1.6. Power Regulation and Protection 156
9.1.7. User Interface Components (LED, Button, Speaker/Amplifier, LCD) 157
9.1.8. System Level Hardware Integration 157
9.1.9. Prototype 158
9.1.10. Senior Design 2 Hardware Plan 159

9.2. Software Testing **160**
9.2.1. MCU Software 161
9.2.2. UI Software 161
9.2.3. Inertial Navigation System Software 163
9.2.4. Infrared Thermal Sensor Software 163
9.2.5. Radar Software 164
9.2.6. BLE Software 165
9.2.7. Senior Design 2 Software Plan 166

CHAPTER 10 ..... *166*

10.1. Budget **166**

10.2. Division of Work **169**

10.3. Milestones **171**

CHAPTER 11 ..... *172*

11.1. Conclusion **172**

Appendix A - References ***1***

Appendix B - Datasheets ***4***

Appendix C - Software Code ***5***

Appendix D – AI Prompts *326*

List of Tables

Table 1: Overall System Specification 9
Table 2: Component Specifications 10
Table 3: Bill of Materials 16
Table 4: Senior Design I Project Documentation Timeline 17
Table 5: Senior Design I Project Design Timeline 17
Table 6: Senior Design II Project Design Timeline 17
Table 7: INS Comparisons 27
Table 8: Barometric sensor comparisons 29
Table 9: Magnetometer sensor comparisons 31
Table 10: UWB product comparison 37
Table 11: Geophone comparison 39
Table 12: Radar Module Comparison Table 43
Table 13: IR Thermal Sensor Comparison Table 51
Table 14: Safety and RF Exposure Management parts comparison 53
Table 15: Pushbutton Comparison Table 58
Table 16: LED Comparison Table 60
Table 17: Buzzer Comparison Table 62
Table 18: Speaker Comparison Table 63
Table 19: LCD Part Comparison 66
Table 20: MCU comparison 71
Table 21: MCU model comparison table 75
Table 22: Power Distribution Table 79
Table 23: 3.3 V Regulator Comparison 84
Table 24: 5 V Regulator Comparison 84
Table 25: Smart Phone Application Communication Comparison 96
Table 26: Android Technology Comparison 99
Table 27: Budget allocation table for sub-systems. 167
Table 28: Bill of Materials 169
Table 29: Division of Work 170
Table 30: Senior Design 1 documentation milestones 171
Table 31: Senior Design 1 design/prototype milestones 171
Table 32: Estimated Senior Design 2 milestones 172

List of Figures

Figure 1: Hardware Block Diagram for 2HADRAD 14
Figure 2: Software Block Diagram for 2HADRAD 14
Figure 3: House of Quality 15
Figure 4: Portable Life Detector Prototype Blueprint. Generated by ChatGPT (OpenAI) March 2026 18
Figure 5: Real Time UI Illustration. Generated by ChatGPT (OpenAI) February 2026 19
Figure 6: Hardware Block Diagram 115
Figure 7: Radar Module PCB Schematic. 116
Figure 8: INS module schematic (original M135 design; MPU9250 + BMP280 substituted on I^2C lines in final build, GNSS/GPS on UART) 117
Figure 9: IR sensor schematic. 118
Figure 10: UWB sensor schematic 119
Figure 11: USS schematic 120
Figure 12: ESP32-WROOM-32E schematic 121
Figure 13: GPIO extender for ESP32-WROOM-32E 122
Figure 14: USB to UART schematic. 123
Figure 15: Schematic for UI: Buttons, Indicators, and Display Connections 125
Figure 16: LM2576 5V Regulator 127
Figure 17: LM2576 3.3V Regulator 127
Figure 18: Power System Carrier Board 128
Figure 19: Use Case Diagram 129
Figure 20: Overall UI Flowchart 131
Figure 21: Sketch of Thermal Map 132
Figure 22: Sketch of Radar Mode 135
Figure 23: Sketch of INS Operating Mode 137
Figure 24: Sketch of Scan Operating Mode. 139
Figure 25: Sketches of Smartphone App UI. 141
Figure 26: iOS App Interface. 142
Figure 27: Sensor subsystem PCB for USS, UWB, IR, INS and MCU 145
Figure 28: 60-pin Radar Breakout Board 146
Figure 29: User Interface PCB 147
Figure 30: Power System Carrier Board 148
Figure 31: LM2576 3.3V Board 149
Figure 32: LM2576 5V Board 150
Figure 33: Testing the IR sensor with the LCD 151

Figure 34: Additional testing done using a human body part. 152
Figure 35: Developed a menu to toggle between sensors..... 152
Figure 36: Testing the USS to ensure it displays when someone is too close. . 152
Figure 37: Components acquired aside from those listed above: Radar module, UWB sensor, INS. 153

CHAPTER 1

1.1. Executive Summary

2HADRAD is a senior design project developed and assembled entirely by electrical and computer engineering students at the University of Central Florida. As technology grows more advanced, it is important to harness it towards saving lives and helping people in dangerous environments. The tools available to first responders working in search and rescue are limited in flexibility and effectiveness. Traditional methods like using animals or individual infrared or acoustic sensors are inadequate. We need multimodal devices that can detect human presence and activity in challenging environments like low light or low visibility. To address this issue, the 2HADRAD is created.

The project is composed of a battery powered handheld device that uses a combination of radar, ultra-wideband (UWB), infrared (IR), inertial navigation system (INS), and barometric sensors. Signals from the sensors are processed in real time using a microcontroller, an ESP32 to detect human presence and activity, such as breathing and heat signatures. Sensor fusion is vital for generating high confidence detections: Radar transmits a chirp to extract I/Q data and conducts onboard digital signal processing to generate a motion/breathing score. The IR sensor checks that same location for 30-40°C human-range heat signatures, providing a thermal-confirmation score to validate radar data. The INS tracks orientation, displacement, and altitude, providing the 3D location where detection occurs, and the UWB performs time-of-flight (ToF) ranging, giving a precise distance estimate to stabilize target location and reduce uncertainty. When a final detection score passes a threshold, the MCU triggers the LED and User interface (UI) alert and logs the victim's 3D coordinates from the fused location data.

2HADRAD is intended for real life search and rescue scenarios in collapsed, dark, smoky, or dusty buildings. The user has total control over the device, the main purpose being to start detection sessions with the device, tracking their own traversal, and potential humans detected by the device's various sensors. A device constraint is a range of 2-4 meters, as this is the range of the IR sensor. Another constraint is that the speed of data being fused and then displayed onto the LCD is limited.

This document outlines the design process of 2HADRAD, including theoretical foundations, enabling technologies, and rationale behind part selection. Standards and constraints guiding the project are discussed, followed by discussion and case studies on the use of Artificial Intelligence Large Language Models (such as ChatGPT). The document further details system architecture, hardware schematics, and software implementation. Subsequently, the document provides information on fabrication, prototyping, and administrative considerations. Final conclusions and recommendations for future improvements

are presented, along with appendices containing technical references, code, and permissions.

CHAPTER 2

2.1. PROJECT BACKGROUND AND MOTIVATION

In the aftermath of disasters the most critical priority is locating and rescuing survivors as quickly as possible. Traditional search and rescue techniques rely heavily on human senses, trained canines, and thermal imaging systems. While these tools have saved countless lives, they each present significant limitations. Canine units require extensive training and can experience fatigue or environmental disorientation, while thermal cameras alone are less effective in smoke-filled spaces, under debris, or when heat signatures are blocked. Similarly, acoustic detection techniques may fail in noisy post-distortion environments. These constraints underline the need for advanced, multimodal technologies that can complement conventional tools and improve the speed, reliability, and safety of rescue operations.

Recent advances in sensing technologies have enabled the development of compact, low power devices capable of detecting human presence and activity in challenging environments. Radar based sensing is particularly promising for search and rescue applications because it is unaffected by lighting conditions, operates effectively in smoke or dust, and offers the capability to detect micro-motions associated with life functions, such as chest movement from breathing. However, radar is not always sufficient, as complex environments and clutter can obscure signals or cause false positives. To address these challenges, integrating radar with complementary sensing methods can significantly enhance the detection of confidence and situational awareness.

This project proposes a handheld device that can detect humans using a combination of radar, UWB, IR, INS, and barometric sensors. Each sensing modality provides a distinct type of information that enhances the reliability and robustness of human detection. Radar serves as the primary detection mechanism, while UWB aids in localization and ranging. IR provides thermal contrast in accessible scenarios, and barometric data supports altitude estimation indoors or underground. If time permits, the addition of seismic sensing allows the system to detect low frequency vibrations from footsteps, tapping, or shifting debris, even when line-of-sight is not available. Depending on mission requirements, the seismic capability can be implemented using a low noise MEMS accelerometer integrated into the handheld unit or a dedicated geophone module connected externally for higher sensitivity.

By integrating these sensing technologies into a single, portable system and applying advanced digital signal processing (DSP) and sensor fusion techniques, this project aims to enhance human detection in complex rescue scenarios. The system is designed to function primarily in line-of-sight conditions but includes a

stretch goal of achieving limited through material detection, expanding its utility in real-world emergency response operations. Ultimately, this approach seeks to support first responders with a tool that is faster, more reliable, and more versatile than current solutions.

2.2. PROJECT FUNCTION

The proposed system is a handheld, battery powered device designed to support first responders in locating survivors during disasters such as earthquakes, building collapses, or fires. The device integrates multiple sensing modalities such as a millimeter-wave radar, UWB, IR, and an INS with barometric sensing, to provide complementary data that improves detection accuracy and situational awareness.

The radar subsystem is the primary detection mechanism, capable of capturing micro-motions associated with human activity. High frequency radar signals penetrate smoke and airborne debris and reflect moving targets, allowing the system to detect subtle motions such as breathing. The signal returned undergoes analog signal conditioning and digitization before being processed with digital signal processing (DSP) algorithms to extract motion patterns and reduce environmental noise and clutter. The UWB subsystem enhances the radar by providing precise ranging and localization capabilities. UWB signals enable accurate distance measurements and tracking of movement, improving spatial awareness in both indoor and outdoor environments. This is particularly valuable in collapsed structures or underground areas where GPS signals are unavailable.

The IR subsystem provides thermal imaging data to highlight human presence where line-of-sight is available. Unlike radar, which operates independently of light and temperature, IR sensing utilizes temperature contrast to confirm the presence and approximate location of a human target, particularly in smoke-filled or visually obstructed environments. The INS and barometric subsystems add spatial context by tracking the device’s movement and altitude changes, enabling responders to maintain positional awareness as they navigate multi-story buildings or subterranean environments. When paired with a global navigation satellite system (GNSS) receiver, the system can further improve positional accuracy by referencing absolute geographic coordinates when satellite signals are available.

Sensor data is fused in real time within the microcontroller unit (MCU), which runs DSP pipelines, sensor calibration routines, and data fusion algorithms such as Mahony or Madgwick filters for attitude estimation, and extended Kalman filters (EKF) for pose estimation and sensor integration. The device assigns confidence scores to detections by cross validating radar, thermal, and ranging data, reducing false positives, and improving detection reliability.

A UI on the handheld device displays detection confidence, relative location, and activity levels, while a companion Android application communicates over Bluetooth Low Energy (BLE) to visualize detections and mark victim locations on

a map. This connectivity enables rapid sharing of situational data among rescue teams. The device is powered by a rechargeable battery system, ensuring portability and continuous operation in field conditions.

While the primary objective is accurate line-of-sight detection, the system is designed with scalability in mind. Future iterations will explore through material detection, extending the device's effectiveness to scenarios involving drywall, wood, or light rubble. By uniting multiple sensing technologies into a single platform, this project aims to provide first responders with a powerful, field deployable tool that enhances search and rescue operations in time-critical environments.

2.3. Current Commercial Technologies and Existing Projects

2.3.1. Portable Sign of Life Identifier (Department of Homeland Security)

The Portable Signs of Life Identifier was a research effort funded by the Department of Homeland Security (DHS) focusing on evaluating compact and portable tools that could detect subtle respiration and heartbeat signals through walls or rubble [7]. The project explored radar and acoustic sensing methods capable of identifying minor chest movements caused by breathing or cardiac activity. The study emphasized portability, rapid deployment, and resilience to environmental interference, aligning closely with the design objectives of the 2HADRAD system.
While the DHS report demonstrated that the radar is capable of successfully detecting vital signs using Doppler radar, it also discussed limitations of the system such as the high-power demand, environmental noise, and false positives triggered by non-human motion. The 2HADRAD project builds on these findings by leveraging more power efficient embedded radar hardware, modern signal processing techniques, and a multi sensor approach to reduce false detections. By integrating radar data with IR and inertial sensors, the system aims to confirm that detected motion correlates with human body temperature, thus improving detection confidence in cluttered environments.

2.3.2. Integration of Multi Sensor Detection Technologies

Multiple academic and industrial studies have explored multi modal life detection by combining radar with other sensing domains such as IR, and acoustic. Multi sensor fusion improves detection reliability by verifying that observed movement correlates with a valid heat or vibration signature. The 2HADRAD system adopts a similar sensor fusion strategy, merging radar data with IR temperature readings and barometric or inertial feedback. This approach enhances both accuracy and interpretability, ensuring that life detection outputs are based on multiple physical indicators rather than radar reflections alone.

Unlike earlier systems relying only on radar, which required external computers for visualization, 2HADRAD integrates a dedicated LCD interface to display immediate feedback such as “Human Detected” or “Area Cleared”. The system’s design philosophy prioritizes speed, portability and user clarity, which are essential in high stress and time sensitive emergency operations. The inclusion real time signal filtering and sensor fusion to differentiate human activity from background motion further improves performance by distinguishing human respiration from background motion, offering a modernized and efficient evolution of earlier prototypes.

2.3.3. Commercial Through Wall Radar Systems: Camero Xaver and Vayyar Imaging

Several commercial radar solutions demonstrate the capabilities of current industry technology. The Camero Xaver 100 and 400 series, developed in Israel, are professional grade through wall imaging radars that use UWB impulse radar to detect movement and breathing through concrete and other solid barriers [11]. These systems offer high accuracy and real time visualization but are limited by large form factors, high cost, and significant power consumption, making them impractical for compact or battery-operated systems.

Similarly, Vayyar Imaging produces radar on chip modules based on frequency modulated continuous wave (FMCW) and UWB techniques that provide 3D imaging and real time presence detection [12]. While Vayyar modules are smaller than the Xaver systems, they rely on proprietary signal processing and high-speed computing, increasing integration complexity for lightweight embedded designs.

By integrating a compact radar module with thermal and inertial sensors on a single custom PCB, the 2HADRAD system delivers the same core capabilities as large commercial life-detection platforms, such as identifying human presence through barriers, but in a far smaller and more power efficient design. Its lightweight, affordable cost architecture makes it well suited for search and rescue kits, field testing, and educational applications, bridging the gap between complex commercial systems and practical embedded solutions.

2.3.4. Infrared Assisted Radar Concepts

Recent academic research has explored combining radar with IR thermography to enhance detection accuracy, particularly in environments with high clutter or minimal motion [13], [14]. These hybrid systems use radar to detect movement and IR sensors to verify heat signatures corresponding to living organisms. Studies have shown that thermal data can significantly reduce false positives caused by moving objects such as fans, machinery, or debris.

The 2HADRAD project adopts this hybrid principle by incorporating an IR temperature sensor (MLX90640) alongside the radar subsystem. The IR module

provides contactless temperature verification, confirming that detected motion corresponds to human body heat (typically 30 - 40 °C). This design choice improves performance in both open and obstructed fields of vision, while maintaining a compact and portable size. By integrating IR, radar, and inertial data within a unified embedded framework, 2HADRAD effectively combines vital sign detection, thermal validation, and environmental context for higher reliability than single sensor designs.

2.3.5. Summary

Both government and commercial studies confirm that detecting human life through walls is achievable but technically complex. Early DHS projects proved radar could measure vital signs, while also revealing the challenges of interference and high-power use. Meanwhile, modern commercial systems like Camero Xaver and Vayyar Imaging demonstrate impressive capability but remain expensive and bulky.

The 2HADRAD project takes these lessons a step further by merging radar, IR, and inertial sensing into a compact embedded design. It delivers a handheld, low power, and affordable life detection system that's both practical and field ready. This multi sensor approach not only enhances accuracy but also makes the technology more accessible to academic researchers and first responders, representing a meaningful step toward the next generation of rapid, portable life detection devices.

2.4. PROJECT OBJECTIVES AND GOALS

The objective of 2HADRAD is to design and build a portable, integrated multi-sensor device that assists rescuers in detecting and localizing human presence during emergency and disaster response. Radar serves as the primary sensing modality, supported by complementary technologies such as UWB, IR, and inertial/barometric sensing. Research and manufacturer demonstrations have shown that radar can detect subtle life signs, such as breathing or even heartbeat, though this project emphasizes broader line-of-sight detection, with limited through-material capability considered as a stretch goal. The device combines sensor fusion, modern signal processing into a unified system, delivering rapid and reliable feedback in real time under field conditions.

2.4.1. GOALS

- Basic:
    - UWB ranging will be integrated to provide approximate distance measurements that extend INS functionality in indoor, GPS denied environments.
    - Confirm human presence by detecting body heat signatures using an IR sensor.

    - Provide relative location and orientation tracking using an INS with barometric feedback.
    - Display detection results on an LCD interface providing immediate visual feedback.
    - Power the entire device with a rechargeable battery subsystem that ensures stable operation.
    - Radar detects motion and respiration
- Advanced:
    - Maintain performance in smoke-filled or low-light environments.
    - Improve relative location tracking precision by using GNSS.
    - Optimize the power delivery for extended operational runtime.
    - Enhance LCD readability with clear formatting and detection indicators.
    - Transmit data from a detection device to an android mobile application through Bluetooth Low Energy
- Stretch:
    - Incorporate a machine learning Layer to improve classification accuracy.
    - Demonstrate radar detection capability through common building materials.
    - Improve the thermal array visualization to produce a clearer, more stable thermal map on the UI.
    - Enhance data transmission to the Android application in real-time and include positional mapping.
    - Integrate a battery monitoring subsystem that displays voltage, charge percentage, and estimated runtime on the LCD.

2.4.2. OBJECTIVES

- Basic:
    - Confirm human presence using an IR temperature sensor under line-of-sight conditions at distances up to ≤ 1 meter.
    - Provide relative location using INS and barometric sensing with an accuracy of ≤ 2 meters.
    - Display detection feedback on an LCD interface within 5 seconds of confirmed detection.
    - Add a red LED to indicate when a human is detected in less than 1 second.
    - Incorporate a button to mark the location of a victim, clear/reset, power the device and navigate the LCD screen.
    - Operate the complete device on a rechargeable subsystem for at least 2 hours of continuous use.
    - Radar detects human distance and respiration within ≤ 2 meters
- Advanced:
    - Demonstrate consistent radar detection performance in cluttered or noisy environments and maintain ≥ 90% detection accuracy.

  - Demonstrate radar-based detection of a human subject within a line-of-sight range of at least 2 meters.
  - Improve fused localization accuracy within ≤ 4 meter when GNSS data is fused with INS outdoors.
  - Enhance LCD readability in field conditions to ≥ 90% legibility under ambient light with clear indicator labels.
  - Transmit data from the handheld device to an android mobile application through BLE within 5 seconds.
  - Improve operational runtime by at least 25% compared to baseline while maintaining ≤ 10 W power draw.
- Stretch:
  - Demonstrate detection of human presence through 1 inch of drywall material in controlled indoor tests with ≥ 80% accuracy.
  - Incorporate a hazard awareness layer that flags 3 potential risks using fused sensor data with ≥ 80% detection accuracy.
  - Add user adjustable controls to UI, add detection modes, heat map visualization and update display within 5 seconds of adjustment.
  - Add real-time battery monitoring to charge percentage and estimated runtime on the UI.
  - Enhance android application with real-time positional mapping with ≤ 2 second latency.

2.4.3. REQUIREMENT SPECIFICATIONS

Parameter	Description	Target Value
Radar detection range	Detect a human subject in LOS	≤ 2 m (LOS)
Detection accuracy	Overall detection performance for radar and IR.	≥ 90%
Average system power	Average electrical power during continuous scanning	≤ 10 W
Battery life	Continuous operation time on rechargeable battery during scans.	≥ 2 hours

Parameter	Description	Target Value
UI response	Time for LCD and speaker/LED to respond after confirmed detection.	≤ 5 second
IR detection	Confirm human presence by detecting body heat signatures.	within ≤ 1m LOS
Location	Track victims' location when detected using INS, GNSS (when applicable) and barometer.	≤ 4 m

Table 1: Overall System Specification

Component	Parameter	Description	Target Value
Radar module	Integration latency	Time from RF transmission to usable baseband output	≤ 200 ms
Radar module	Detection range	Detect human motion under LOS conditions	≥ 2 m
IR sensor	Human heat confirmation	Detects body heat signatures to confirm human presence	≤ 1 m
IR sensor	Interface and conditioning	Digital bus, add small series resistors for edge damping.	I2C/SPI with 22-47Ω series, proper decoupling
INS with barometer	Relative localization	Dead reckoned displacement over short paths	≤ 2m error over 10 m
GNSS (advanced)	Fusion anchor	Absolute position when available	≤ 4 m
UWB (advanced)	Ranging precision	Sensor fusion with radar to extend range	≥ 3 meters
LCD	Feedback latency	Time from confirmed detection to display	≤ 5 s

Component	Parameter	Description	Target Value
LED	Turn on latency	Visual alert	≤ 1s
Wireless	Telemetry latency	From detection device to mobile application	≤ 10 s
MCU	Logic and supply	Core voltage and primary input/output for links and wireless	3.3 V, processor class at least 240 MHz
MCU	Interfaces	Serial links for INS, IR sensor, barometer, ranging for radio and telemetry.	at least 2 universal asynchronous receiver-transmitter links, 1 I2C and 1 SPI interface bus
MCU	Processing/signal processing	Filtering, detection logic, sensor fusion, and UI updates	Display update within 5 s and LED within 1 s

Table 2: Component Specifications

2.5. RELATED STANDARDS

Our team anticipates conforming to the following national and/or international standards:

- FCC Part 15: including but not limited to subparts A, B, C and F
- IEEE 802.15.4: UWB communications standard
- IEC 62133 / UL 2054: Li-ion battery safety
- ETSI EN 305 550: European equivalent to FCC Part 15 to cover 57-64 GHz short-range devices
- ESTI EN 302 065: Covers 3.1 - 10.6 GHz band.
- USB-IF USC Type-C and Power Delivery standards
- IEC 62368-1/UL 60950 for DC input safety
- IEEE 52-2019: Standard letter designations for radar-frequency bands
- IEC 62366: Application of using engineering for medical devices.
- API 35 for new android applications
- Build OWASP MASVS (Mobile application security verification standard)
- Follow WCAG 2.1 level AA

2.6. PROJECT BLOCK DIAGRAM

Figure 3 illustrates the signal flow and major subsystems of the proposed handheld human detection device. The design integrates radar, UWB, IR, inertial and barometric sensors into a unified system, all managed by a central MCU. The figure represents a single PCB implementation where sensing, processing, UI, and power management are consolidated into a compact and portable platform.

At the core of the system is the TI IWR6843AOP radar module, which transmits and receives a 60 GHz frequency modulated continuous wave (FMCW) chirp. Reflected signals return as baseband I/Q data, which pass through an analog to digital converter (ADC), a hardware accelerator, and a C674x DSP core for low level signal processing. The radar has the capability of detecting human micro-motions such as breathing or small body movements, even in environments with low visibility caused by dust, or smoke. Radar outputs are then sent to the ARM R4F control core and passed to the MCU for higher level processing, fusion, and decision making.

The sensors subsystem complements the radar by integrating additional modalities that improve detection accuracy and situational awareness. To expand the radar's sensing capabilities and strengthen overall system performance, the device incorporates an UWB subsystem, which is specifically chosen for its ability to complement radar data by improving ranging precision and providing additional spatial information. Previous research and industry solutions have demonstrated that combining UWB with radar improves localization and reduces uncertainty in complex environments. In this project, UWB measurements will help refine detection zones and can aid in limited through material sensing, extending the radar's effectiveness in cluttered or partially obstructed scenarios. The IR sensor provides thermal contrast, confirming the presence of humans when line-of-sight is available and highlighting heat signatures even in visually obstructed environments. The INS, combined with a barometric pressure sensor, enables inertial navigation by tracking device orientation, motion, and elevation changes, critical for maintaining positional awareness in multi-story buildings or underground areas.

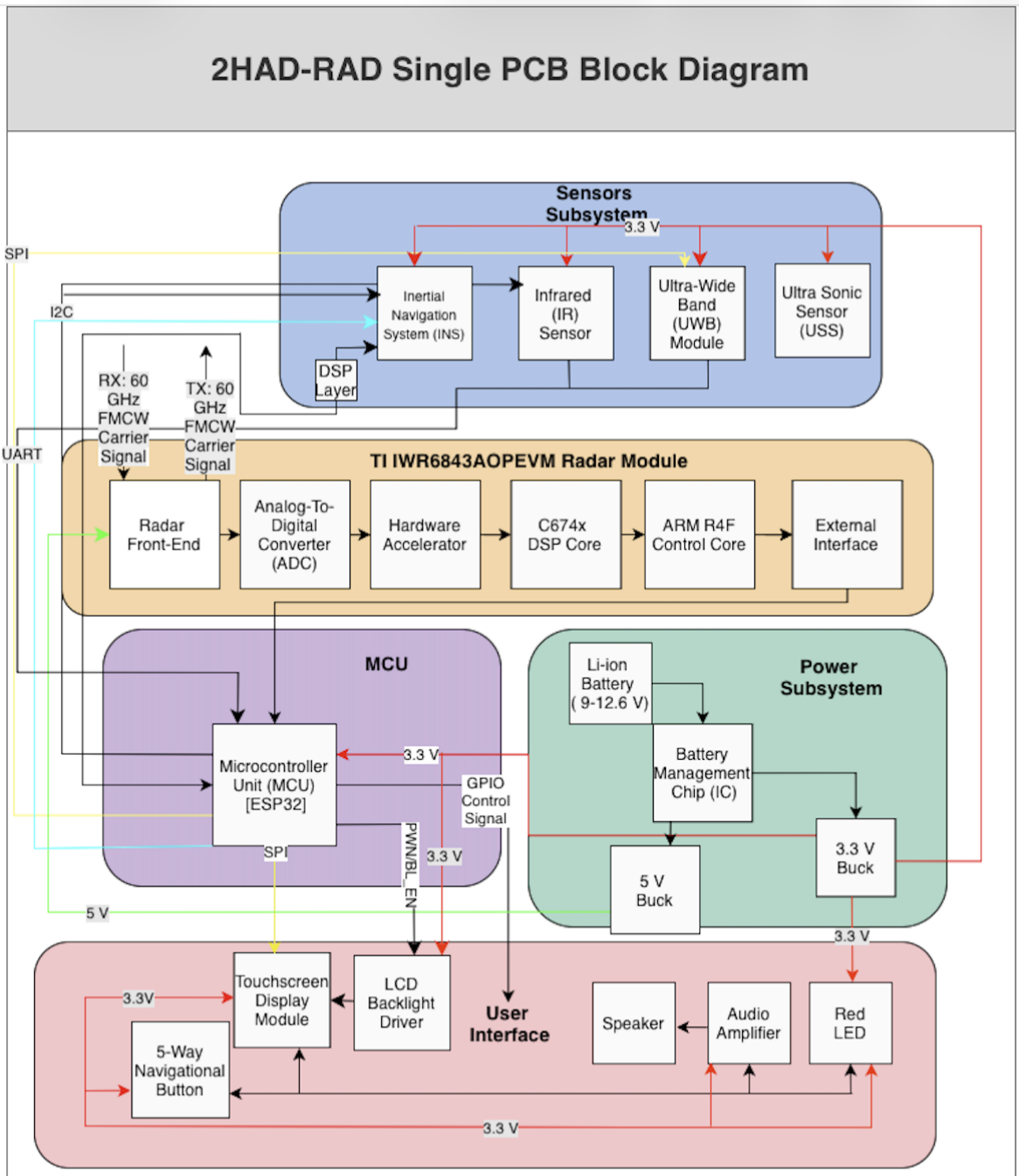
All sensor data converge at the MCU, which serves as the central processing hub. The MCU performs multiple roles: it coordinates data acquisition from each sensor, applies DSP algorithms and filtering, and executes sensor-fusion logic to combine inputs into a unified detection decision. Algorithms such as Mahony or Madgwick filters are used for attitude estimation, while extended Kalman filters (EKF) may be implemented for fusing IMU, UWB, and barometric data into a coherent position estimate. Detection of confidence scores is generated by cross validating radar, thermal, and ranging data to reduce false positives and improve decision reliability.

The UI subsystem provides intuitive feedback to the operator. A display presents key metrics such as detection of confidence, relative location, and sensor activity. LEDs and a speaker provide quick visual and audible alerts when potential human presence is detected. Additionally, BLE connectivity links the device to an Android application, which visualizes detection data, marks victim locations, and enables rapid communication among response teams.

The power subsystem is built around a rechargeable lithium-ion battery, supplying regulated 3.3 V power to all modules through a buck converter and low noise linear regulators. The power circuit includes charging management and protection features to ensure safe and continuous operation in the field. Finally, the division

of labor table shows how subsystem responsibilities are distributed among team members according to their expertise. This structure allows specialized focus on radar processing, sensor integration, power delivery, UI design, and communication modules, while enabling collaborative development and integration of the full system.

Figure 2 (software block diagram) further illustrates the flow of information within the device. The MCU centralizes data from all sensing subsystems and processes it through layers of filtering, calibration, and fusion. LED indicators respond to radar detection results, and combined sensor confidence outputs. Meanwhile, localization and movement data from the INS, UWB, and barometric sensors feed into the mapping interface, which updates the user's display and Android application in real time. Together, these hardware and software layers enable a coordinated, multi-modal detection system designed to improve the speed, accuracy, and reliability of search and rescue operations.



Division of Labor		
Subsystem	**Primary Lead**	**Secondary Support**
Power	Ryan Andersen (EE)	Bailey Hitt (EE)
Sensors	Shannon Hankinson (EE)	Bailey Hitt (EE)
Radar	Ryan Andersen (EE)	Shannon Hankinson (EE)
MCU	Shannon Hankinson (EE)	Joshua D'Souza (CpE)
User Interface	Bailey Hitt (EE)	Shannon Hankinson (EE)

Figure 1: Hardware Block Diagram for 2HADRAD

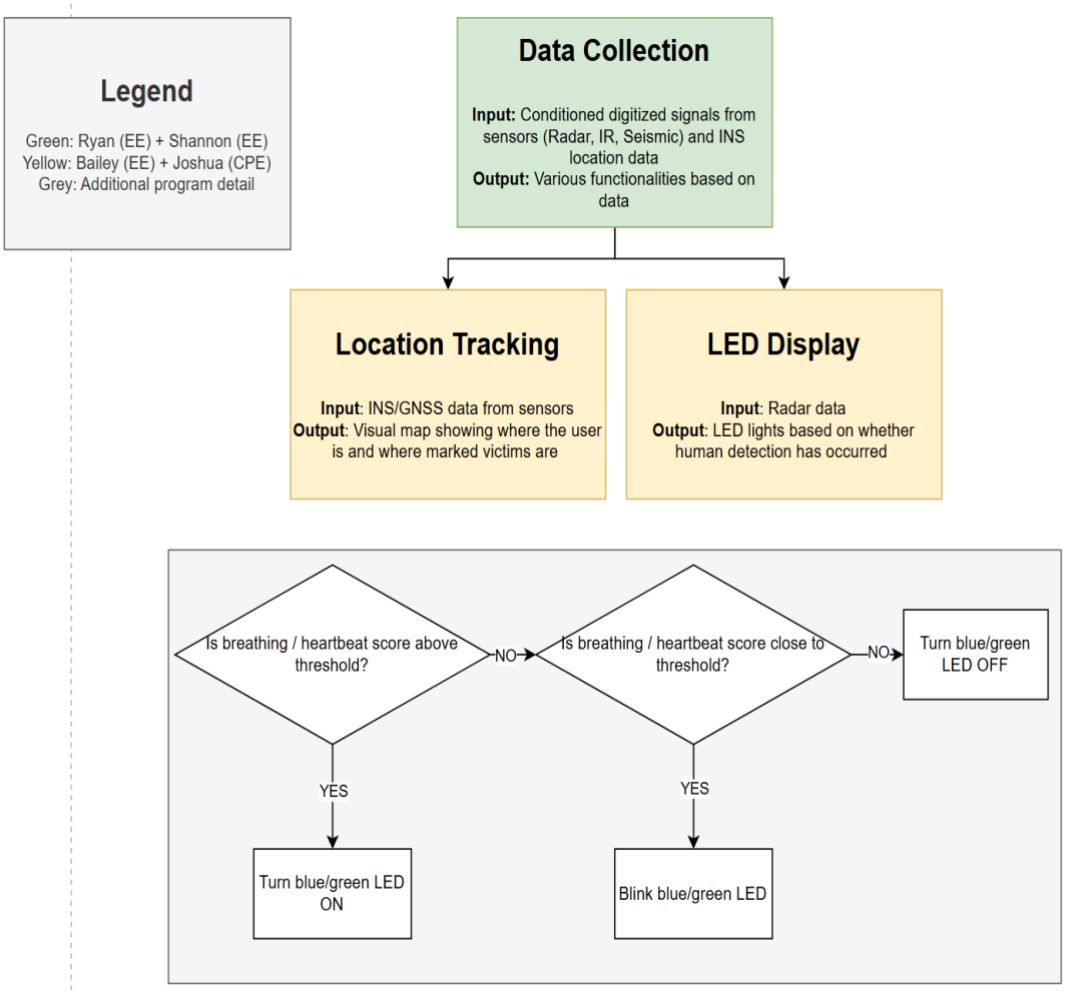


Figure 2: Software Block Diagram for 2HADRAD

2.7. HOUSE OF QUALITY

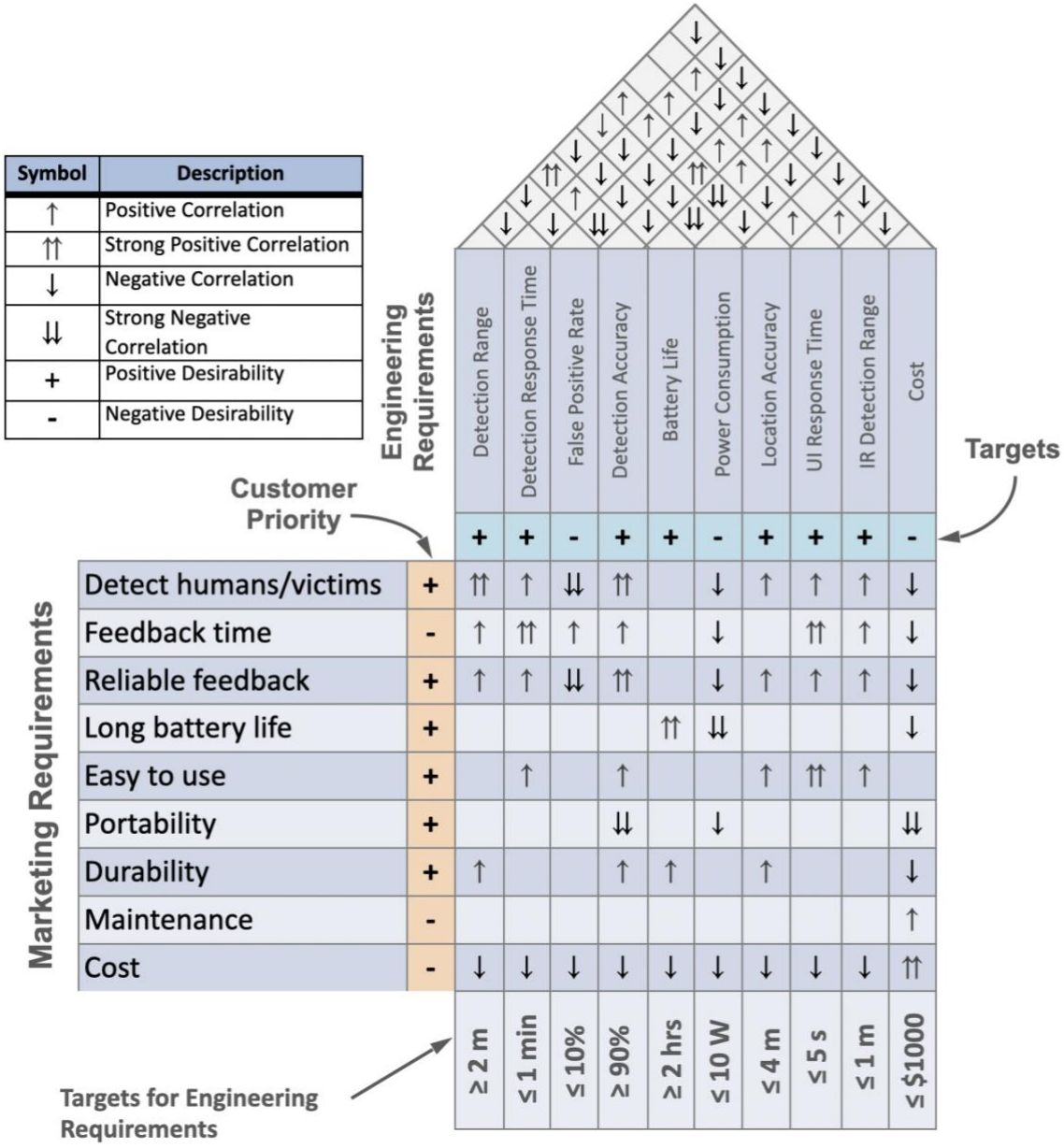


Figure 3: House of Quality

2.8. PROJECT BUDGET AND FINANCING

Our team will self-fund this project, so it is in our best interest to keep the total budget as conservative as possible. Here is the estimated total bill of materials (BOM) as can be foreseen currently:

Item	Total Cost
Radar front end module	200
Radar PCB	88
Other PCBs	280
Amp / Speaker	16
LCD	20
INS	120
UWB x2	60
Battery Pack	40
Infrared Sensor	80
PETG Printing filament	40
Additional Componenets (Digikey)	50
Total	994

Table 3: Bill of Materials

2.9. INITIAL PROJECT MILESTONES

Our team formed in the Spring semester of 2025 when we began brainstorming. However, our serious efforts did not begin until the start of the Fall semester of 2025. Therefore, the timeline begins at the start of the Fall 2025 semester.

Item	Start Date	Anticipated End Date	Duration
Project Brainstorming	8/19/2025	8/26/2025	0 Weeks
Project Scope Finalized	8/26/2025	9/6/2025	1 Week
Individual Research	8/26/2025	9/6/2025	2 Weeks
Initial Project Proposal	8/26/2025	9/6/2025	1.5 Weeks
30-Page Milestone	9/6/2025	9/27/2025	3 Weeks
60-Page Milestone	9/27/2025	10/17/2025	3 Weeks
90-Page Milestone	10/17/2025	11/7/2025	3 Weeks
120-Page Milestone	11/7/2025	11/25/2025	3 Weeks
Final Draft	11/25/2025	12/3/2025	1 Week

Table 4: Senior Design I Project Documentation Timeline

Item	Start Date	Anticipated End Date	Duration
Individual System Design	9/5/2025	10/2/2025	4 Weeks
Individual System Testing	10/2/2025	10/30/2025	4 Weeks
Breadboard Prototyping	10/30/2025	11/21/2025	3 Weeks
PCB Design/Ordering	11/21/2025	12/12/2025	3 Weeks

Table 5: Senior Design I Project Design Timeline

Item	Start Date	Anticipated End Date	Duration
PCB Testing	1/9/2026	1/23/2026	2 Weeks
System Integration/Testing	1/23/2026	2/16/2026	4 Weeks
Practice Project Demo	2/16/2026	2/27/2026	2 Weeks
Finalize Documentation	2/27/2026	3/13/2026	2 Weeks
Practice Final Presentation	3/13/2026	3/20/2026	1 Week
Final Presentation	TBD	TBD	TBD

Table 6: Senior Design II Project Design Timeline

2.10. PROTOTYPE

To visualize the final form of the system, a conceptual rendering of the physical device and user interface was created. The design depicts a rugged, handheld unit roughly 8 inches wide, resembling a compact tablet with protective side handles for secure operation in field environments.

The front of the device integrates a central color LCD display that presents the system interface, including radar visualization and system status information. Below the display, a multi-directional navigation pad with an OK selection button

allows users to control the interface while wearing gloves or operating in difficult conditions.

A status LED indicator on the front panel provides immediate visual feedback for system alerts and operational states. The enclosure is designed with a durable, textured housing to withstand demanding environments while maintaining portability. This form factor enables first responders and search-and-rescue personnel to quickly operate the system in low-visibility scenarios, where traditional visual detection methods are limited.

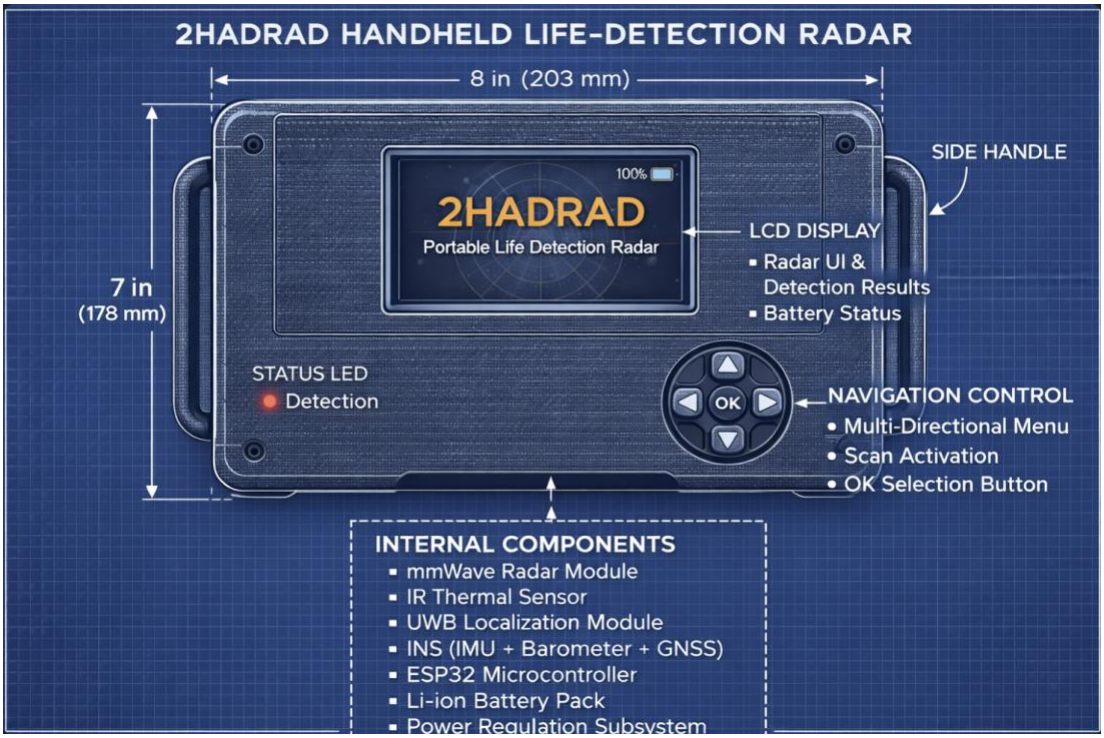


Figure 4: Portable Life Detector Prototype Blueprint. Generated by ChatGPT (OpenAI) March 2026

Furthermore, the prototype illustration for the UI can be seen in the figure below. The interface highlights real time alters, thermal detection overlays and mapping functions to provide rescuers with immediate and actionable information.

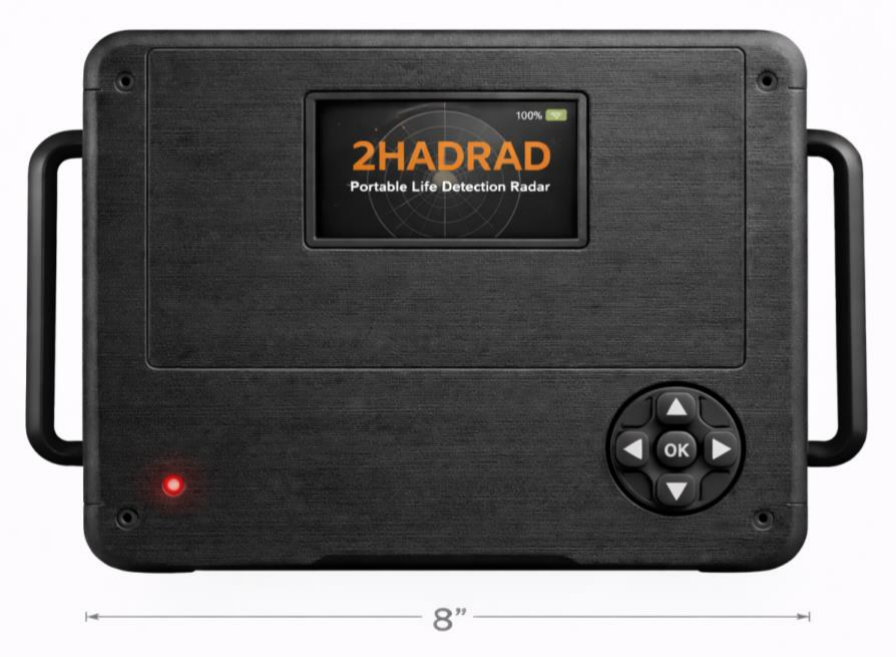


Figure 5: Real Time UI Illustration. Generated by ChatGPT (OpenAI) February 2026

CHAPTER 3

3.1. Location/Tracking

Several sensing technologies can contribute to pinpointing locations, each with unique capabilities and limitations. In this project, we researched various methods of locating victims without the use of wireless communications. In this project, we evaluate Inertial Measurement Units (IMUs), INS, UWB sensor, and IR and Radar modules as potential solutions.

3.1.1. Inertial Measurement Unit

An IMU has two separate configurations: a six-axis and a nine-axis. The six-axis IMU combines a three-axis accelerometer and a three-axis gyroscope. The nine-axis uses the same two components but adds a three-axis magnetometer. The additional component provides a reference to magnetic north, providing more accurate heading and attitude data to help correct sensor drift over time.

The six-axis configuration measures linear acceleration and gravity using the accelerometer while the gyroscope measures angular rate. Together these components are ideal for measuring tilt, detecting motion, and orientation changes. In this project, we want to implement a way to mark the location of a victim. The six-axis will provide a general location of the victim due to the orientation drifting over time, the yaw will not be precise, and the stored heading may be relative. However, it still delivers strong performance for many basic functionalities needed in search and rescue situations. It is sufficient for tasks like

marking a location, logging path segments and relative orientation awareness. The six-axis IMU draws less power, offers a simpler calibration, and is immune to magnetic field corruption.

The nine-axis configuration adds the measurement of Earth's magnetic field vector. This allows the magnetometer to correct the drift periodically the gyroscope introduces. An academic study "Statistical Sensor Fusion of a 9-DoF MEMS IMU for Indoor Navigation" shows that using accelerometer, gyroscope, and magnetometer together in a robust unaltered calibration framework reduces orientation error significantly in environments with magnetic distortion, improving orientation accuracy by 80-90%, compared to gyroscope and accelerometer alone. However, magnetometers require careful calibration and are vulnerable to some local disturbances such as steel framing, machinery, or power lines.

3.1.2. Inertial Navigation System

An INS combines motion sensing with computational processing to provide continuous estimates of position, velocity, and orientation. It consists of an IMU typically a six-axis or nine-axis sensor which is integrated with a processor that executes the strapdown mechanization equations. The IMU's three-axis accelerometer measures linear acceleration, allowing the system to detect motion, tilt, and displacement, while the three-axis gyroscope measures angular rate to track rotations and directional changes. A three-axis magnetometer, when included, references the Earth's magnetic field to correct yaw drift, thereby ensuring consistent heading even during prolonged operation. Together, these sensors form a self-contained, nine-axis configuration capable of maintaining relative motion tracking without any external signals.

To enhance long term accuracy, many INS modules also include a GNSS receiver such as GPS, Galileo, or BeiDou. The GNSS provides absolute position fixes, while the INS interpolates user movement between these updates using its high-frequency inertial data. When the GNSS signal becomes unavailable, such as inside buildings, tunnels, or debris, the INS continues to estimate motion autonomously. It relies on internal sensors and algorithms, such as the Kalman filter, to model and correct sensor drift, bias, and scaling errors. In the context of this project, the INS plays a critical role in providing relative location tracking for search and rescue operations. As rescuers move through a disaster zone, the INS continuously logs their motion path and orientation. When a victim is found, the rescuer can precisely mark their location relative to the rescuer, even if satellite coverage is weak or completely lost, by marking it in local coordinates. Once the rescuer returns to an area with GNSS visibility, these relative positions can be correlated with absolute coordinates to generate a map of victim locations. This hybrid approach ensures that every detected victim can be precisely revisited or relayed to other team members, even in GNSS denied environments.

By integrating the high update rate and drift-corrected attitude of the INS with periodic GNSS fixes, the system offers a dependable dead-reckoning navigation solution that ensures accuracy even during transitions between outdoor and

indoor environments. In short, the INS ensures that responders always know where they have been, where they found victims, and how to guide others back, without relying solely on external infrastructure.

3.1.3. Ultra-Wideband (UWB)

UWB is a short-range radio frequency (RF) technology that transmits extremely brief pulses over a wide frequency spectrum, typically between 3 and 10 GHz. UWB's wide bandwidth allows for precise ToF distance measurements, often within a range of 10–30 cm. This makes UWB highly suitable for determining relative distances and movements between devices. Unlike Wi-Fi or Bluetooth, UWB's broad spectral spread allows it to resist multipath interference from walls, debris, and reflective materials, ensuring stable signal integrity in cluttered environments.

While conventional UWB positioning systems rely on fixed anchors to calculate position, such infrastructure is often unavailable in disaster zones. Instead, this project considers UWB as a complementary sensing layer to the INS, where it could enhance relative localization among rescuers or between multiple handheld devices. For example, when a victim is located, the location is marked using the INS, UWB could help refine that relative position by providing ranging data between rescuers' devices in a local network. This cooperative positioning approach could improve mapping consistency when multiple responders are searching in different areas, especially when GNSS signals are unavailable.

Though UWB alone cannot serve as a global navigation method without fixed anchors, integrating its short-range precision with the INS's continuous trajectory tracking offers a promising pathway for improved relative mapping, spatial awareness, and team coordination in the field.

3.1.4. IR and Radar Module

IR systems detect infrared radiation emitted by warm bodies against cooler backgrounds. There are two types, passive and active IR. Passive IR or thermal cameras detect temperature differences and generate images or heat maps. Active IR or structured IR beams project IR light and measure reflections to detect presence or movement. If the IR is calibrated with distance, it can provide both direction and approximate distance. However, this is limited to line of sight only and heat sources can create false detections.

Radar transmits radio waves, measures reflections from objects, and detects motion from doppler shifts or locates targets through ToF and range resolution. Specialized bio-radar systems can detect chest motion through obstacles. Radar modules can estimate the range and angle of a detected person, which translates into a location estimate relative to the radar. Radar may be able to pinpoint a location of a victim through light obstacles but not through heavy or metallic barriers. Although both modules may be useful to aid an IMU or INS, it is not ideal as is.

3.1.5. Barometric Pressure Sensor

Barometric pressure sensors measure atmospheric pressure and convert it into height above sea level using the barometric formula. This conversion enables a system to estimate changes in elevation. In search and rescue operations, this capability is crucial for providing vertical context. For instance, it helps distinguish between floors in a collapsed building or identify when a rescuer ascends or descends stairwells. When combined with the INS and GNSS data, the barometer completes a three-dimensional tracking framework, enabling rescuers not only to log horizontal movement but also to determine the altitude of each detected victim location.

In the proposed system, when a rescuer marks the location of a victim using a handheld device, the barometer measures the ambient pressure at that precise point. This reading, combined with INS and GNSS data, generates a 3D coordinate. This coordinate provides both the relative (INS based) and absolute (GNSS based) positions, along with the vertical dimension obtained from the barometer. Such integration ensures that responders can later interpret whether a victim is located on a specific floor or depth level, crucial for prioritizing rescue routes and coordinating multi-level operations.

For this project, the barometric sensor must balance accuracy, stability, and low power consumption to suit portable, battery powered hardware. High-resolution devices like the Bosch BMP390 and Infineon DPS310 can achieve sub-meter altitude precision while drawing only a few microamps, making them ideal for continuous, low-drift monitoring. Their compact size and I2C/SPI interfaces simplify direct integration with the INS subsystems. Conversely, lower cost options like the Bosch BMP280 still provide meter level accuracy, which may be adequate for rough altitude tracking but could limit effectiveness in multi-story or subterranean search scenarios where precision vertical mapping is required. In essence, the barometric pressure sensor transforms the INS from a planar tracker into a comprehensive 3D localization system. This transformation provides rescuers with not only a map of the victims' locations but also their relative height or depth in relation to the rescuer's position.

3.1.6. Magnetometers

Magnetometers add an essential function to location tracking systems by measuring the Earth's magnetic field and providing absolute heading information. While accelerometers and gyroscopes can describe motion and orientation, they suffer from drift over time, especially in yaw, which directly affects navigation accuracy. A three-axis magnetometer corrects this issue by acting as a compass reference, which is particularly important indoors or in urban areas where GNSS reception is unreliable. For a search and rescue tool, this means a rescuer can maintain correct orientation while mapping a path or marking the location of victims. The choice of magnetometer must consider sensitivity, noise, and how well the device tolerates interference from nearby metal structures or electronics. Compact and efficient devices such as the Bosch BMM150 are widely used for

basic applications, while higher-end parts like the MEMSIC MMC5983MA include features such as set/reset coils that help cancel residual magnetic offsets. In the context of this project, selecting a magnetometer with strong accuracy and resistance to interference, such as the LIS3MDL or MMC5983MA, would provide the most reliable heading data to support both victim location and team navigation in challenging environments.

3.2. Location Part Selection

For this project, we are selecting a 6 or 9-axis INS module, for which we plan to integrate a barometric pressure sensor. The INS provides continuous motion tracking through its internal IMU, while the barometer adds precise altitude and floor-level detection. The integrated GNSS enables absolute positioning outdoors, complementing the INS and preventing long term drift. This combination allows the handheld device to operate reliably in both indoor and outdoor environments, providing rescuers with accurate location information even when a victim cannot be immediately rescued.

3.2.1. Inertial Navigation System

3.2.1.1. M5Stack M135 Module

The M5Stack M135 is a ready to use positioning board that integrates a u-blox NEO-M9N-00B GNSS module, a BMI270 six-axis IMU, a BMM150 three-axis magnetometer, and a BMP280 barometric pressure sensor. This INS module is all in one and supports multi-constellation GNSS: GPS, GLONASS, Galileo, BeiDou, QZSS, and can update navigation up to 25 Hz. Its GNSS accuracy is about 1.5 meters under good conditions. The IMU provides selectable accelerometer ranges up to +/- 156.96 m/s^2 and gyroscope range up to +/- 2000º/s. The magnetometer has a field resolution of approximately 0.2µT. The barometer covers pressure ranging from 300 to 1100 hPa, usable for detecting altitude or floor changes.

The module also includes a backup battery to retain data during power loss. The M135 comes with a small dual in-line package (DIP) switch block on its PCB. These switches provide direct hardware level control without needing to reflash firmware. The pins that the GNSS and sensors connect to may be reassigned if needed. The GNSS module defaults to a 9600 baud rate, but this may be increased using the DIP for higher throughput GNSS data logging. The IMU, magnetometer, and barometer sit on the same I2C bus. This module is specified to operate at altitudes up to 80,000 m, sustain accelerations of about 39.2 m/s^2, and function at velocities as high as 500 m/s. The module also requires a fixed supply voltage of 3.3 V and has physical dimensions of 54 x 54 x 13.1 mm, with a total weight of 14.4 g. However, this module does not include any algorithms onboard, but we can apply them ourselves on an external MCU.

3.2.1.2. STMicroelectronics Teseo-VIC3DA Module

The ST Teseo-VIC3DA is a compact dead-reckoning GNSS module that integrates an AM330LHH six-axis IMU with ST’s Teseo-DRAW firmware for on

module sensor fusion. It supports multi-constellation GNSS tracking including GPS, GLONASS, Galileo, BeiDou, and QZSS. Under open sky conditions, its GNSS accuracy is approximately 1.5 meters circular error probable. This module can output dead-reckoning fixes up to 30 Hz, while the raw GNSS update rate is 1 Hz. The integrated IMU offers selectable accelerometer ranges up to +/- 156.96 m/s^2 and the gyroscope ranges up to +/- 2000º/s. the VIC3DA does not include a magnetometer or barometric sensor on-chip, so external sensors would be required to achieve a nine-axis solution or altitude aiding capabilities.

The device also incorporates features such as map-matching feedback and an odometer input for wheel ticks, which can be used to enhance dead-reckoning accuracy when the GNSS signals are obstructed. Firmware updates are supported, and a temperature compensated crystal oscillator maintains frequency stability over -40º C to +85º C. This module requires a supply voltage of 3 to 3.6V and has a very low standby current consumption. The physical dimensions are 16 x 12.2 x 2.42 mm in a 24-pin LCC package. While it delivers integrated sensor fusion through ST's algorithms, users who need a magnetometer or barometer must supplement the module with external devices to achieve a complete INS configuration.

3.2.1.3. U-blox NEO-M9V

The u-blox NEO-M9V is a dead-reckoning GNSS module designed for high-efficiency positioning in both automotive and embedded applications. It integrates a six-axis IMU directly into the module and employs an onboard Kalman filter algorithm to blend IMU and GNSS data for smoother navigation in weak-signal conditions. It supports multi-constellation GNSS reception (GPS, Galileo, GLONASS, BeiDou) and provides navigation solutions at update rates up to 25 Hz. Under nominal conditions, horizontal positioning accuracy is typically within 1.5 meters.

The onboard IMU supports accelerations up to +/- 156.96 m/s^2 and angular rates up to +/-2000°/s, making it suitable for tracking dynamic motion. Unlike the M135, the NEO-M9V does not integrate a magnetometer or barometer, though the module is designed to accept external I2C connected sensors for these functions. Operating limits allow for high dynamic use cases, including maximum altitudes of 50,000 meters and velocities up to 500 m/s, consistent with GNSS industry constraints.

The module requires a supply voltage of 2.7 to 3.6 V and has a very low power consumption of approximately 60 to 90 mW in continuous tracking mode, making it ideal for battery-powered field devices. Its compact footprint measures 12.2 x 16.0 x 2.4 mm in a standard u-blox NEO package. While the NEO-M9V delivers robust dead-reckoning and multi-band GNSS capabilities at a low power draw, the lack of onboard magnetometer and barometric pressure sensing means external components are required for a complete nine-axis INS solution. Although, the GNSS can determine altitude in outdoor areas.

3.2.1.4. U-blox ZED-F9R-03B

The u-blox ZED-F9R-03B is a high-performance, dual-band GNSS dead-reckoning module that combines a six-axis IMU, a multi-constellation dual-band GNSS receiver, and an onboard Kalman filter to deliver continuous navigation solutions. It supports GPS L1/L2, Galileo E1/E5b, BeiDou B1I/B2I, and GLONASS L1 signals, enabling robust positioning even in dense urban or obstructed environments. The module achieves a navigation output rate of up to 30 Hz, with real-time kinematic support for centimeter-level accuracy when correction data are provided.

The integrated IMU supports accelerometer ranges up to +/-156.96 m/s² and gyroscope ranges up to +/-2000°/s, allowing the device to maintain high-precision dead-reckoning during GNSS outages. Like the NEO-M9V, the ZED-F9R does not include a magnetometer or barometer but supports the addition of external I2C sensors for nine-axis capability and altitude aiding. Operational limits are consistent with other GNSS modules, supporting altitudes up to 50,000 m, accelerations up to 39.2 m/s^2, and velocities up to 500 m/s. However, this module is only capable of determining the altitude in outdoor conditions due to no barometer including in the package.

The ZED-F9R-03B requires a 2.7 to 3.6 V supply and has a typical power consumption of about 85 mA at 3.0 V at about during continuous tracking. Its footprint is 17 × 22 × 2.4 mm in the ZED LGA form factor, offering more PCB real estate than the NEO series but with higher performance. Although it has a higher cost relative to the VIC3DA and NEO-M9V, the ZED-F9R provides dual-band robustness, RTK capability, and superior accuracy, making it the strongest candidate for advanced field deployments where precision is critical.

3.2.1.5. Custom INS Stack

The ATGM336H GPS+BDS dual-mode module is a compact GNSS receiver designed for embedded positioning and navigation applications. It supports simultaneous reception of GPS (L1) and BeiDou (B1) satellite constellations**,** improving positioning reliability and reducing time-to-first-fix in partially obstructed environments. The module can track up to 32 satellites simultaneously and typically achieves positioning accuracy of about 2.5 m CEP under open-sky conditions. It provides a navigation update rate of up to 10 Hz, allowing responsive location tracking for mobile systems. Communication is handled through a UART interface using standard NMEA messages, making it easily compatible with microcontrollers such as Arduino and ESP32 platforms.

The ATGM336H typically operates from a 3.3–5 V supply depending on the carrier board and consumes approximately 25–35 mA during continuous tracking. Cold start acquisition is typically around 35 seconds, while hot start reacquisition can occur in under 2 seconds. With its compact size and integrated antenna options on many modules, the ATGM336H provides a cost-effective alternative to common GNSS modules such as the NEO-6M and NEO-M8N for embedded navigation applications.

The MPU9250 (GY-9250) 9-axis motion sensor integrates a three-axis accelerometer, three-axis gyroscope, and three-axis magnetometer into a single MEMS device, enabling full nine-degree-of-freedom motion tracking. The accelerometer supports selectable ranges of ±2 g to ±16 g, while the gyroscope provides rotational measurement ranges from ±250 to ±2000°/s. The integrated AK8963 magnetometer measures magnetic field strength up to ±4800 μT, enabling electronic compass functionality and heading estimation. Together, these sensors allow measurement of roll, pitch, and yaw orientation, making the device suitable for inertial navigation and motion tracking applications.

The MPU9250 supports both I²C and SPI communication interfaces and operates from a 2.4–3.6 V supply with typical current consumption of approximately 3.5 mA during active operation. The device also includes a digital motion processor (DMP) capable of performing onboard sensor fusion, reducing the computational load on the host microcontroller. Its compact design and integrated sensing capabilities make it a widely used solution for embedded motion sensing systems.

The BMP280 atmospheric pressure sensor is a high-precision MEMS barometric sensor designed to measure atmospheric pressure and temperature for altitude estimation and environmental monitoring. It measures pressure in the range of 300–1100 hPa, corresponding to altitudes from approximately –500 m to 9000 m above sea level. The device features 24-bit pressure resolution with noise levels as low as 0.2 Pa, allowing altitude resolution of roughly 1–2 cm under ideal conditions. An integrated temperature sensor provides compensation to improve measurement accuracy across varying environmental conditions.

The BMP280 operates from a 1.71–3.6 V supply and consumes approximately 2.7 μA during measurement and 0.1 μA in sleep mode, making it suitable for low-power embedded systems. It supports both I²C and SPI communication, enabling straightforward integration with microcontrollers such as the ESP32. Due to its small footprint, low power consumption, and high precision, the BMP280 is commonly used in altitude sensing, indoor navigation, weather monitoring, and drone flight control systems.

Module	Sensors on board	GNSS	Power	Supply	Size(mm)	Cost
u-blox ZED-F9R-03B	3-D IMU (includes sensor fusion)	Dual band multi-constellation, RTK/PPP-RTK support, up to 30 Hz	About 0.255 W	2.7-3.6 V	17x22x2.4	$149-$164
u-blox NEO-M9V-20B	Six-axis IMU	L1 multi-constellation, Automotive/untethered dead-reckoning, up to 50 Hz (default 1Hz)	About 0.147 W	2.7-3.6V	16x12.2x2.4	$35

Module	Sensors on board	GNSS	Power	Supply	Size(mm)	Cost
ST Teseo-VIC3DA	6-axis IMU	L1 multi-constellation with automotive dead-reckoning FW	0.161 W	3.3 V	16x12.2x2.42	$19-$21
M5Stack M135	Barometer, 6-axis IMU, magnetometer	L1 multi-constellation, up to 25 Hz	0.111 W	3 V	54x54x13.1	$45
Custom	9-axis IMU (accelerometer, gyroscope, magnetometer) + barometer	GPS + BeiDou dual-mode positioning, up to ~10 Hz navigation update	~0.18 W (combined typical)	3.3 V	40 x 60	$37

Table 7: INS Comparisons

3.2.1.6. INS Selection Justification

The custom inertial navigation subsystem used in this project is composed of three discrete sensing components: the ATGM336H GNSS receiver, the MPU9250 nine-axis inertial measurement unit, and the BMP280 barometric pressure sensor. This modular architecture was selected to provide full navigation capability while maintaining flexibility, cost efficiency, and compatibility with the system's custom hardware platform. The ATGM336H provides satellite-based positioning using GPS and BeiDou constellations, while the MPU9250 integrates a three-axis accelerometer, three-axis gyroscope, and three-axis magnetometer for orientation and motion sensing. The BMP280 provides high-resolution atmospheric pressure measurements that can be converted into altitude estimates, completing the sensing suite required for three-dimensional positioning.

Unlike fully integrated navigation modules, the custom sensor stack separates the GNSS receiver from the inertial sensors, allowing each component to be independently selected and optimized for the system requirements. The MPU9250 communicates with the ESP32 microcontroller through an I²C or SPI interface, while the ATGM336H outputs standard NMEA navigation messages over UART. The BMP280 operates on the same I²C bus as the IMU, enabling efficient communication with minimal wiring complexity. This distributed design simplifies debugging, allows individual sensors to be replaced or upgraded if needed, and integrates cleanly with the project's custom PCB architecture.

Functionally, this sensor stack enables the system to estimate a victim's three-dimensional position within the search area. The GNSS receiver provides absolute geographic coordinates when satellite signals are available, while the IMU supplies continuous motion and orientation data that can support short-term dead-reckoning. The magnetometer assists with heading estimation, and the barometer provides vertical height information. When a detection event occurs, the system records the position estimate using the fused sensor data, allowing

responders to mark and revisit locations even if GNSS reception becomes temporarily degraded.

In summary, the custom ATGM336H–MPU9250–BMP280 sensor stack provides a cost-effective and adaptable navigation solution while still delivering the full set of measurements required for inertial navigation and three-dimensional localization. Although it requires external sensor fusion software on the ESP32 microcontroller, this approach offers greater design flexibility and significantly lower cost compared to integrated INS modules, making it well suited for a portable search-and-rescue detection platform.

3.2.2. Barometer

3.2.2.1. Bosch BMP390

The Bosch BMP390 is one of the most precise barometric pressure sensors currently available for compact designs. It comes in a 2.0 x 2.0 x 0.8 mm package and offers a relative accuracy of about +/- 3 Pa, which corresponds to roughly +/- 0.25 m in altitude. Absolute accuracy is rated around +/- 0.5 hPa over a wide temperature range, while its temperature coefficient of offset (TCO) is only +/- 0.62 Pa/K, which helps maintain stable readings when the environment changes. The device operates over a pressure range of 300 to 1250 hPa and can sample at up to 200 Hz. At a 1 Hz update rate, it consumes only about 3.2 µA, with a sleep current near 1.4 µA, making it well suited for low power portable devices. While it is slightly more expensive than older parts, its noise performance and accuracy make it a strong candidate for applications where detecting floor changes or subtle altitude shifts is important.

3.2.2.2. Infineon DPS310

The Infineon DPS310 is a digital barometric pressure sensor designed for low power consumption and strong thermal stability. It uses a capacitive sensing element, which makes it less sensitive to sudden temperature changes, such as when a user moves between indoor and outdoor environments. The device is packaged at 2.0 x 2.5 x 1.0 mm and provides a relative accuracy of +/- 0.06 hPa (about +/- 0.5 m) and an absolute accuracy of +/- 1 hPa. It supports both I2C and SPI interfaces and can achieve measurement rates up to around 157 Hz, depending on the oversampling settings. Typical current is only a few microamps in low frequency operation, making it an attractive choice for battery powered applications. One advantage of the DPS310 is its internal 32 sample FIFO buffer, which reduces the load on the microcontroller. With its combination of precision, stability, and low cost, the DPS310 represents a practical option for adding altitude awareness to an INS system.

3.2.2.3. Bosch BMP280

The Bosch BMP280 is packaged in a 2.0 x 2.5 x 0.95 mm LGA, it provides a relative accuracy of about +/- 0.12 hPa (+/- 1 m altitude) and a typical absolute accuracy of +/- 1 hPa. Its TCO is higher than that of the BMP390, around 1.5

Pa/K, which means its readings can drift more in changing temperatures. Power requirements are very low, drawing about 2.7 µA at a 1 Hz update rate, with standby currents below 0.3 µA. Interfaces include I2C and SPI, both operating at relatively high speeds. Although its accuracy does not match newer sensors, it is inexpensive, widely available, and sufficient for many applications where floor level precision is not required.

3.2.2.4. ST LPS22HB

The ST LPS22HB is a compact barometer, measuring at 2.0 x 2.0 x 0.76 mm, that emphasizes robustness and flexibility. It covers a pressure range of 260 to 1260 hPa and achieves a typical relative accuracy of +/- 0.1 hPa. Power consumption depends on the mode, ranging from about 3 to 12 µA at 1 Hz. In addition to its low power features, the LPS22HB includes built-in FIFO storage and programmable interrupts, allowing it to signal the microcontroller when thresholds are crossed, reducing the need for constant polling. It is also designed to withstand high over pressure and extreme shock levels up to 22,000 g, making it suitable for rugged environments.

Sensor	Relative Accuracy	Absolute Accuracy	Pressure Range	Power	Output Data Rate	Supply Voltage	Size (mm)
Bosch BMP390	+/-0.25 m	+/-4.2 m	300-1250 hPa	0.0106 mW	200 Hz	1.65–3.6 V	2.0 x 2.0 x 0.8
Infineon DPS310	+/-0.5 m	+/-8 m	300-1200 hPa	0.0165 mW	157 Hz	1.7-3.6 V	2.0 x 2.5 x 1.0
Bosch BMP280	+/-1 m	+/-8.3 m	300-1100 hPa	0.0089 mW	157 Hz	1.71-3.6 V	2.0 x 2.5 x 0.95
ST LPS22HB	+/-0.8 m	+/-0.8-8.3 m	260-1260 hPa	0.0099-0.0396 mW	75 Hz	1.7-3.6 V	2.0 x 2.0 x 0.76

Table 8: Barometric sensor comparisons.

3.2.2.5. Barometer Selection Justification

While the M5Stack M135 already integrates a Bosch BMP280 barometric pressure sensor, additional research was conducted to evaluate higher precision and lower power alternatives, such as the Bosch BMP390, Infineon DPS310, and ST LPS22HB. These standalone sensors were reviewed to evaluate all our options and identify potential improvements in altitude accuracy, thermal stability, and integration complexity. The newer BMP390 and DPS310, for instance, offer sub-meter precision and advanced temperature compensation, making them attractive options for fine grained vertical mapping in multiple story search scenarios. Since the integrated BMP280 in the M135 offers sufficient performance to meet the project’s resolution requirements, effectively distinguishing floor-level changes and elevation shifts, it was decided to retain the onboard component. This choice minimizes the need for additional wiring, conserves power and simplifies firmware development. It still enables reliable

three-dimensional localization when combined with INS and GNSS data. The evaluation revealed that while higher-end sensors could slightly enhance vertical accuracy, the integrated barometer provides the most practical balance of performance, efficiency, and integration simplicity for this prototype stage.

3.2.3. Magnetometer

3.2.3.1. MEMSIC MMC5983MA

The MMC5983MA is a high performance three-axis magnetometer that uses anisotropic magneto resistive (AMR) technology. It offers an 18-bit resolution and extremely low RMS noise of about 0.4 mG, making it one of the most precise magnetometers available in this class. The measurement range is +/- 8 Gauss, and the device supports output data rates up to 1000 Hz. A unique feature of the MMC5983MA is its built-in set/reset function, which helps cancel residual magnetic offsets that can build up in metal-rich environments. It operates from 1.8 to 3.6 V and supports both I2C and SPI interfaces. Its small footprint (3 × 3 mm) and low current draw make it feasible for embedded systems, though its cost is somewhat higher than simpler devices.

3.2.3.2. ST LIS3MD

The LIS3MDL from STMicroelectronics is a widely adopted three-axis magnetometer with selectable full-scale ranges of +/- 4, +/- 8, +/- 12, or +/- 16 Gauss. It provides 16-bit output data and supports both I2C and SPI interfaces, with output data rates up to 1000 Hz. Power consumption depends on the mode, ranging from around 40 µA in low power configurations to 270 µA in ultra-high-resolution mode at 20 Hz. The sensor is rated for operation between −40 ºC and +85 °C and comes in a compact 3 x 3 mm package. It also includes built-in self-test and interrupt features, making it easier to integrate into systems that require reliable event driven heading measurements. The LIS3MDL represents a balanced option in terms of accuracy, flexibility, and cost, and it is well supported in software libraries and hardware platforms.

3.2.3.3. QST QMC5883L

The QMC5883L is a low-cost magnetometer that uses AMR technology and provides a three-axis magnetic field measurement with a typical heading accuracy of 1 to 2° after calibration. Its operating voltage range is 2.16 to 3.6 V, and it supports I2C communication. The maximum output data rate is about 200 Hz, which is sufficient for many basic navigation tasks but lower than higher-end devices. The small size and inexpensive price make it attractive for student projects and prototypes, but performance can vary depending on the manufacturer, so calibration is essential for reliable results.

3.2.3.4. Bosch BMM150

The BMM150 is a compact three-axis magnetometer that is often paired with Bosch IMUs in sensor fusion applications. It measures magnetic fields up to +/- 1300 µT in the X and Y axes and up to +/- 2500 µT in the Z axis. Typical heading

accuracy is about +/- 2.5° under good calibration conditions. The sensor supports I2C and SPI interfaces and offers both forced and normal operating modes, with a maximum output data rate of 30 Hz. Current consumption is around 170 µA at 10 Hz in low-power mode, and the package measures only 2 x 2 mm. Although its maximum data rate is lower than other options, its integration within the Bosch sensor ecosystem makes it easy to pair with IMUs such as the BMI270. This makes it a practical choice for projects where power efficiency and compact integration are more important than high-speed sampling.

Sensor	Range	Noise/Res.	ODR	Power	Supply Voltage	Heading Accuracy
MEMSIC MMC5983MA	+/- 8 G	0.4 mG RMS	1000 Hz	2.2 mW	1.8-3.6 V	+/- 0.5º (well calibrated)
ST LIS3MDL	+/- 4/8/12/16 G	3.2 mG typical	1000 Hz	0.12-0.89 mW	1.9 – 3.6 V	+/- 1-2º
QST QMC5883L	+/- 8-20 G	0.5-2 mG	200 Hz	0.33 mW	2.16-3.6 V	+/- 1-2º (needs calibrated)
Bosch BMM150	+/- 1300 µT (XY), +/- 2500 µT (Z)	0.3 µT res.	30 Hz	0.56 mW	1.62-3.6V	+/- 2.5º typical

Table 9: Magnetometer sensor comparisons

3.2.3.5. Magnetometer Selection Justification

Similarly, while the M5Stack M135 features a Bosch BMM150 three-axis magnetometer, a comparative review was conducted to explore alternative magnetometers that could improve heading accuracy and interference tolerance. This review also provided insights into our available options. Alternatives such as the ST LIS3MDL, MEMSIC MMC5983MA, and QMC5883L were analyzed for factors including magnetic sensitivity, noise, and offset compensation. High performance devices like the MMC5983MA demonstrated superior resolution and active offset cancellation features, while the LIS3MDL provided broader range settings and strong library support for embedded platforms. However, these benefits come at the cost of additional wiring, calibration steps, and power consumption. The BMM150 is already internally aligned with the M135's IMU on the same bus, ensuring consistent axis orientation, compact integration, and minimal setup complexity. The onboard magnetometer provides sufficient accuracy for heading correction and drift mitigation in fusion with the accelerometer and gyroscope. Therefore, the team concluded that retaining the integrated BMM150 provided the most efficient and practical solution, fulfilling the system's requirements for orientation tracking, drift correction, and absolute heading without introducing additional hardware dependencies.

3.3. Life Detection Modules

In search-and-rescue applications, the ability to detect human presence through common building materials such as drywall, wood, and concrete is critical for

locating victims who may not be visible to thermal or optical sensors. Traditional sensing solutions, such as radar modules and IR cameras, provide valuable information but can be limited in their penetration depth or sensitivity to stationary individuals. To enhance the overall system, this research investigates alternative modules and sensors, particularly those that use UWB or related technologies, that are capable of transmitting signals through obstacles and detecting subtle human motion, such as respiration. The primary objective of this section is to evaluate candidate components that complement existing radar and IR systems, providing redundancy and increased reliability when searching for victims in visually obstructed environments.

3.3.1. Ultra-Wideband (UWB) Modules

Beyond localization, UWB also offers valuable capabilities for life detection, particularly when fused with millimeter-wave radar. UWB modules transmit pulses across a wide frequency range, usually between 3.1 and 10.6 GHz. These pulses can penetrate non-metallic materials like drywall, wood, and light concrete. Lower frequency components within this band achieve greater penetration depth, while higher frequencies offer finer spatial resolution. This dual behavior allows UWB to estimate the distance and presence of a target behind obstacles, even when direct line of sight is blocked.

When combined with radar in a sensor fusion framework, each technology compensates for the other's limitations. Radar offers high-resolution Doppler and micro-motion signatures, including the subtle chest motion caused by breathing. On the other hand, UWB provides depth and material penetration data. By merging both outputs through algorithms such as an Extended Kalman Filter or weighted fusion model, the system can both detect life signs and estimate victim location through barriers.

This hybrid UWB–radar subsystem is designed to feed its data directly into the INS-based mapping system, where the detected signal's relative position and range can be logged as a 3D pin. The result is a multi-modal detection and localization framework capable of confirming human presence and recording precise spatial context, even in GNSS-denied or visually obstructed environments.

UWB modules are also well-suited for field deployment due to their compact size, energy efficiency (operating at 100–150 mW active), and already supported robust communication protocols (IEEE 802.15.4z). With proper signal processing techniques like clutter suppression and motion extraction, UWB significantly enhances the radar's ability to differentiate between living targets and noise. This improvement leads to enhanced sensitivity and reliability in the handheld device's life detection capabilities. In this project, a DW3000 chip serves as the mobile Ping initiator on the ESP32 device while a Qorvo DWM3000EVB module serves as the Pong responder on the Arduino Uno R4 WiFi.

3.3.2. Low Frequency RF Modules

Unlike higher-frequency systems (24 GHz or 60 GHz radar), low-frequency RF modules take advantage of improved penetration properties in dense materials. At these lower frequencies, attenuation in concrete and wood is significantly reduced, enabling detection through walls greater than 30 cm thick in some cases. By operating in sub-GHz bands, the tradeoff is spatial resolution. As bandwidths are typically narrow, limiting localization precision to meters rather than centimeters. They provide an alternative when UWB signal strength is insufficient to penetrate through thicker barriers. The modules are often larger due to the size of the antenna. The power usage is usually between 50 to 100 mW. These systems may not retrieve small motions like respiration as effectively as UWB, but their penetration depth is a significant advantage.

3.3.3. Seismic Sensors

While RF-based systems such as radar and UWB provide strong potential for through-wall detection, they can be limited when faced with dense, rebar-reinforced concrete or when operating in environments with high electromagnetic noise. In such scenarios, a seismic sensing modality serves as a highly effective complement by directly coupling to structural surfaces and capturing mechanical vibrations that propagate through walls, debris, or flooring. These vibrations, generated by human activity such as knocking, shifting limbs, or even subtle body motion, travel through materials that often block radio frequency signals.

A promising approach for integrating seismic sensing into the handheld device is to embed a MEMS accelerometer within a spring-loaded contact pad on the housing. The spring maintains continuous pressure against the surface, ensuring strong mechanical coupling and consistent transmission of structural vibrations into the sensor. This design greatly improves the sensor’s sensitivity compared to a PCB mounted accelerometer alone and reduces interference from operator induced noise. As vibrations travel through the structure, they are mechanically transmitted through the spring assembly to the MEMS sensor, which converts them into acceleration signals. These signals can then be digitally filtered to isolate the low frequency components associated with human movement. Advanced signal processing, such as band-pass filtering and fast Fourier transforms (FFT), can extract spectral signatures of footsteps, tapping, or shifting debris, which are then fused with radar and UWB data to improve detection confidence and reduce false positives.

While MEMS accelerometers generally offer lower sensitivity to ultra-low frequency vibrations compared to geophones, this spring-mounted approach significantly narrows that gap and maintains the advantages of compact size, low power consumption, and direct digital interfacing. It also supports rapid deployment in field conditions: responders can quickly press the device against a wall or surface and begin scanning for signs of life without needing to deploy separate hardware.

Alternatively, a geophone remains a strong option for scenarios requiring maximum sensitivity to very low frequency vibrations. In such a configuration, the geophone would be deployed as a separate, externally placed module, positioned in contact with the ground or structural surface and connected to the handheld system via a shielded cable. This stationary setup minimizes noise from operator movement and preserves the geophone's exceptional low frequency performance, though it sacrifices portability and integration simplicity.

Ultimately, the choice between these two seismic sensing strategies will depend on project priorities and testing outcomes. A spring mounted MEMS accelerometer offers a fully integrated, compact solution ideal for rapid scanning and earthquake response scenarios, while an external geophone module provides unmatched sensitivity for long duration or stationary monitoring. Both represent viable pathways to enhance the system's capability to detect human presence in RF challenging environments.

3.3.4. Radar Module

The radar module serves as the core sensing element of our system, enabling the detection of subtle physiological motions such as respiration and heartbeat potentially even when line-of-sight is obstructed by obstacles like drywall, debris, or smoke. Millimeter-wave (mmWave) radar operates in the 60–81 GHz range, using short wavelengths that provide high spatial resolution, strong material penetration, and robustness to environmental conditions such as dust, darkness, and temperature changes, characteristics that are essential in search and rescue operations where traditional sensing modalities like cameras fail. By transmitting a continuous frequency modulated signal and analyzing its reflections, the radar can determine both the range and velocity of objects and possibly detect micro motions on the order of millimeters caused by human breathing and heartbeat. These capabilities enable it to distinguish living individuals from static objects and provide continuous monitoring even in low visibility environments. Because radar-based sensing is passive, noncontact, and unaffected by ambient light, it is particularly suited to detecting trapped or unconscious victims where other sensors might miss critical signs of life. For these reasons, the radar module is the cornerstone of our system architecture, and careful evaluation of available radar technologies was a critical step in our design process.

3.3.5. Infrared Thermal Sensor

An IR thermal sensor detects heat as opposed to visible light, which allows it to sense the presence of warm objects such as people, animals, or machinery. It works by measuring the IR radiation naturally emitted from surfaces and converting that energy into temperature readings. Each measurement represents the thermal intensity of a small region in the sensor's field of view, which can then be analyzed to determine relative heat patterns. In essence, an IR sensor functions as a thermometer array at a distance, eliminating the need for contact, allowing it to capture real time temperature data that provides insight into environmental and biological activity.

We selected an IR thermal sensor because it enhances our system's ability to detect human presence based on body heat rather than motion alone. While radar provides distance and movement information, the IR module contributes temperature-based verification, improving the reliability of detection in scenarios where movement may be minimal. In this project, the IR sensor will be programmed to recognize temperature ranges corresponding to human body heat, typically between 30°C and 40°C. When readings fall within that range, the system will display a "Human Detected" message on the LCD screen. The stretch goal for this subsystem involves using the same thermal data to generate a color-based heat map visualization on the display. The sensor communicates with the ESP32WROOM microcontroller using the I2C protocol, ensuring simple wiring, low power operation, and straightforward data acquisition. By incorporating this module, the system gains an additional sensing layer that supports both accuracy and environmental adaptability in field applications.

3.3.6. CO-2 Sensor

A CO_2 sensor is particularly effective for life detection because it can identify human respiration by detecting the exhaled carbon dioxide in the air. Every breath a person takes releases carbon dioxide (CO_2) at concentrations significantly higher than the ambient level, resulting in a measurable increase in CO_2 levels when someone is present, even behind barriers like rubble or walls. In rescue or monitoring systems, non-dispersive infrared (NDIR) CO_2 sensors are commonly employed due to their accuracy and selectivity. These sensors detect the IR absorption characteristics of CO_2 molecules. This makes them valuable for detecting trapped survivors in confined spaces because carbon dioxide tends to accumulate when ventilation is restricted. However, their detection range is often constrained by factors such as airflow, diffusion rate, and sensor sensitivity. Despite this, a CO_2 sensor offers direct evidence of metabolic activity, making it a reliable indicator of life in enclosed or low-visibility environments.

3.4. Life Detection Module Part Selection

The proposed life detection system prioritizes the radar module as the primary sensing technology because it has a proven ability to detect minute physiological motions, such as breathing or heartbeat, even through non-metallic materials like drywall or debris. Radar's non-contact nature and resilience to lighting and temperature variations make it a dependable core sensor for continuous monitoring. To enhance this, an IR sensor is integrated to capture thermal signatures indicative of human presence. This sensor provides valuable surface level temperature data that complements radar readings, particularly when line-of-sight is available. The UWB module is chosen for sensor fusion with the radar system, as research suggests that combining UWB's precise ToF measurement with radar's Doppler sensitivity significantly improves detection accuracy and material penetration performance. UWB's low power and wide spectrum signals also help distinguish living subjects from static objects by providing more precise resolution of distance and motion. Lastly, a seismic sensor is set as a stretch goal, which could be incorporated later to detect low-frequency vibrations or

knocks transmitted through solid surfaces. While not crucial for the initial prototype, incorporating this feature in future iterations could enhance detection capabilities to ground coupled signals. This improvement would lead to more accurate survivability assessments in complex rubble environments. This multi-sensor architecture strikes a harmonious balance between penetration depth, precision, and environmental adaptability, providing a solid foundation for the development of next generation life detection systems.

3.4.1. Ultra-Wideband Modules

3.4.1.1. DWM3000EVB

The ultra-wideband ranging subsystem is implemented using a DW3000 ultra-wideband transceiver chip as the mobile Ping initiator on the ESP32-based 2HAD-RAD device and a Qorvo DWM3000EVB module as the Pong responder on the Arduino Uno R4 WiFi. This configuration enables reliable single-sided two-way ranging (SS-TWR) while supporting rapid prototyping and final custom PCB integration.

3.4.1.2. ULM3-STM32 + DWM3000

The ULM3-STM32 + DWM3000 stack combines a STM32 microcontroller with the DWM3000 UWB transceiver in a single embedded platform. Supporting IEEE 802.15.4z protocols, the stack delivers about +/- 5 cm ranging accuracy in the 6.25 to 8.25 GHz band with an average active power draw of 120 mW. This integration reduces system complexity by eliminating the need for an external processing unit, making the design more compact and suitable for handheld applications. It also includes an onboard OLED display and offers a detection range up to 40 m in an open area when communicating at 6.8 Mbps.

3.4.1.3. DWM1000

The DWM1000 is a fully integrated UWB transceiver module developed by Qorvo. It is based on the DW1000 IC, which complies with the IEEE 802.15.4-2011 UWB standard (HRP PHY) and supports ToF ranging and location services. The module integrates the RF front end, crystal oscillator, and antenna in a compact package measuring approximately 23 mm x 13 mm x 2.9 mm. According to the datasheet, the device supports data rates of 110 kbps, 850 kbps, and 6.8 Mbps, with line-of-sight (LOS) accuracy ranging from +/- 10 cm under favorable conditions. Its operating frequency spans from 3.5 to 6.5 GHz and it is powered from a 2.8 to 3.6 V supply. Active transmit current is listed as 46 mA at 3.3 V, while receive current is 50 mA, translating to approximately 150 to 165 mW of active power consumption.

3.4.1.4. DWM3000

The Qorvo DWM3000 is a surface mounted UWB transceiver module that integrates the DW3110 chipset and complies with the IEEE 802.15.4z standard. It is designed to support secure ranging and localization through ToF

measurements, with LOS accuracy on the order of +/- 10 cm as specified in the product documentation. The UWB module transmits extremely short pulses, which spread across a wide frequency band and allow the signals to penetrate materials like drywall with relatively low loss. The module operates within 6.5 to 8 GHz with supported data rates of 850 kbps and 6.8 Mbps, enabling both high-speed and low power communication modes. It is supplied from a 2.5 to 3.6 V source and achieves typical active currents in the tens of milliamperes range, corresponding to power consumption near 100 to 150 mW during transmission or reception. With a compact footprint of 13 x 13 x 2.9 mm, the DWM3000 provides an integration ready form factor that can be directly interfaced to an external microcontroller through standard SPI communication. This combination of size, accuracy, and compliance with modern UWB standards makes the module suitable for embedded applications requiring precise ranging or human detection in multipurpose sensing systems.

Module	Frequency	Power Consumption	LOS	Data Rate	Supply Voltage	Cost
DWM3000EVB	6.5 to 8 GHz	100mW	+/- 10 cm	850 kbps / 6.8 Mbps	2.5 to 3.6 V	$29.50
ULM3-STM32 + DWM3000	6 to 8.5 GHz	120 mW active	+/- 5 cm	6.8 Mbps	External 3.7 to 5 V	$88.06
DWM1000	3.5 to 6.5 GHz	150 to 165 mW	+/- 10 cm	110 kbps/850 kbps/6.8 Mbps	2.8 to 3.6 V	$25-$30
DWM3000	6.5 to 8 GHz	100 to 150 mW	+/- 10 cm	850 kbps / 6.8 Mbps	2.5 to 3.6 V	$25

Table 10: UWB product comparison

3.4.1.5. UWB Selection Justification

The DWM3000EVB was selected as the primary development and deployment platform for the UWB ranging nodes due to its rapid prototyping capability and hardware compatibility. As an evaluation board with an Arduino shield form factor, it interfaces directly with the Arduino Uno R4 WiFi without requiring custom breakout adapters or fine-pitch soldering. This significantly reduces integration time and allows the team to focus on ranging performance, sensor fusion, and localization algorithms rather than PCB rework.

For the final handheld system, the DW3000 transceiver IC will be integrated directly onto the main sensor PCB as the mobile Ping initiator. This approach reduces size, power overhead, and system complexity while enabling tighter electrical integration with the radar, INS, and control electronics. Using the DWM3000EVB during development ensures validated firmware and ranging performance before migrating to the custom PCB implementation.

Functionally, both the EVB and IC utilize the same DW3000 UWB transceiver, operating in the 6.5–8 GHz band and providing centimeter-level ranging accuracy using time-of-flight measurements. The EVB's onboard antenna, standard headers, and documented SPI interface enable rapid validation of ranging performance, while the custom PCB implementation allows optimization for enclosure constraints, power distribution, and electromagnetic compatibility in the final handheld device.

In the operational concept, the handheld unit carries the custom DW3000-based board as the mobile Ping initiator, while the DWM3000EVB serves as the Pong responder on the Arduino Uno R4 WiFi. This architecture enables accurate relative positioning indoors and in GPS-denied environments, allowing detected victim locations to be logged and revisited.

The UWB localization system complements radar detection and INS tracking by providing spatial context. Radar confirms human presence, INS tracks responder motion, and UWB ranging provides position correction and team awareness. Together, this sensor fusion approach supports reliable victim localization and improved responder coordination in complex rescue environments.

3.4.2. Geophones

3.4.2.1. Geophone SM-24

The SM-24 is a vertical geophone element, widely used in seismic and structural monitoring. It is designed to detect vertical vibrations starting at about 10 Hz and can pick up motion up to a few hundred hertz. It is sensitive enough to detect small ground motions and is very consistent across units. It has been labeled as a "workhorse" sensor in both academic research and industry. For the application of this project, the SM-24 would excel at capturing knocks, footsteps, or vibrations traveling through floors or rubble as these often fall within the 10 to 200 Hz range. It does not require any external power source and is encased in aluminum which makes it ideal for portable integration. However, it is not ideal for extremely slow motions below 10 Hz, such as subtle body movements or shifts. It performs best when it is firmly mounted on a stable surface with tilt angles under 10 degrees to avoid damping errors.

3.4.2.2. SM4 Horizontal Geophone

The SM4 variant is oriented for horizontal motion sensing rather than vertical. It's often used for shear wave or lateral vibration detection in structural and seismic systems. According to manufacturer catalogs from Sercel and similar suppliers, the SM-4 share core coil and damping parameters to SM-24, delivering comparable sensitivity around 28-30 V/m/s but tuned for horizontal axes. It begins responding at 10 Hz, maintaining strong performance up to 160 through 190 Hz before other unwanted responses start to show up. This configuration is valuable in environments with multi-directional disturbances such as building collapses or uneven terrain. Housed in a compact, weather resistant case, it

requires secure orthogonal mounting to the vertical unit, and like its sibling, it falters below 10 Hz for very low velocity events.

3.4.2.3. EG-4.5-II Geophone

The EG-4.5-II is a low frequency vertical geophone engineered for enhanced sensitivity to subtle seismic events, featuring a natural frequency of 4.5 Hz that expands its usable bandwidth between 4.5 and 200 Hz. In comparison to the previous choices, this geophone offers a sensitivity of about 40 V/m/s and has a moving coil design that minimizes internal damping, low distortion, and high signal to noise ratios. This model could be ideal for detecting micro-motions from breathing, crawling, or distant footsteps. In our project, this component would provide a critical low pass extension to the SM-24's capabilities. It remains compact and portable but is slightly larger than the SM series.

Specification	SM-24	SM-4	EG-4.5-II
Orientation	Vertical	Horizontal	Vertical and Horizontal available
Natural Frequency	10 Hz	10 Hz	4.5 Hz
Sensitivity	28.8 V/m/s	28.8 V/m/s	28.8 V/m/s
Damping (oc)	0.25	0.25	0.6
Max Tilt Angle	10º	25º	15º
Bandwidth	10-240 Hz	10-180 Hz	4.5-140 Hz

Table 11: Geophone comparison

3.4.2.4. Geophone Selection Justification

For seismic sensing, we opted for the SM-24 vertical geophone due to its ruggedness, and suitability for our primary structural vibration band. In a search and rescue scenario, the SM-24 is highly effective at capturing knocks, footsteps, shifts in debris, and other structural signals that travel through floors, walls, and rubble. This practical and low-power modality serves as a reliable complement to UWB/radar detections. While it is not optimized for sub-10 Hz micro-motions, its strengths align with our immediate objectives: detecting impulsive and low-audio events through contact coupling, integrating those detections into our INS pin-drop workflow, and guiding responders back to likely victim locations. Due to its mechanical integration requirements and secondary priority, this component is currently considered a stretch goal. It will be incorporated if time and resources permit during later development stages.

3.4.3. Radar Modules

3.4.3.1. Infineon BGT60TR13C

The Infineon BGT60TR13C is a compact 60 GHz FMCW radar sensor designed for operation in the unlicensed ISM band under FCC Part 15, offering an attractive balance of size, power, and baseline sensing capability for embedded applications. It features one transmit and three receive channels and includes an

antenna-in-package (AiP), which reduces the need for external antenna design and tuning compared to bare die radar solutions. Its small 6.5 x 5 mm footprint makes it suited for highly space constrained systems, and its typical active power consumption of 0.5–0.6 W fits comfortably within our system's power budget. The device supports the high sampling rates necessary to achieve sub 200 ms end-to-end latency and operates across a wide temperature range (−40 °C to +85 °C), exceeding our environmental requirements.

However, the BGT60TR13C outputs only raw intermediate frequency (IF) data over an SPI interface and does not include on-chip digital signal processing. As a result, all range fast Fourier transforms, doppler processing, and micro doppler extraction must be implemented on an external processor such as the ESP32. This shifts significant computational burden to the host MCU, potentially increasing firmware complexity and impacting real-time performance, especially when additional tasks such as wireless communication and user-interface updates are considered. Although the built-in AiP simplifies integration, antenna orientation, enclosure design, and PCB layout still strongly influence the resulting beam pattern and detection range. Achieving consistent vital sign detection through drywall at 2 m, one of our stretched design goals, would require careful antenna positioning, extensive signal processing optimization, and potentially external amplification or filtering stages.

From a development perspective, Infineon's software support is limited primarily to low level drivers and evaluation tools. Application-level algorithms for respiration and heartbeat detection would need to be built from the ground up, increasing development time and risk compared to more integrated solutions. While the BGT60TR13C is appealing for its small size, low power, and low unit cost, these benefits are offset by the additional design complexity it introduces. It is best suited for designs where custom signal processing pipelines and antenna systems are acceptable or where minimizing cost is prioritized over rapid prototyping and system level simplicity.

3.4.3.2. Infineon BGT60ATR24C

The Infineon BGT60ATR24C is a 60 GHz FMCW radar sensor designed for applications that require higher spatial resolution and multi target detection. Compared to the BGT60TR13C, it provides two transmit and four receive channels, enabling beamforming and angular estimation that improve localization accuracy. This capability is particularly useful for distinguishing multiple targets and resolving subtle motion signatures. The device's compact package is well suited for embedded designs, and its wide automotive-grade temperature range (−40 °C to +125 °C) exceeds the system's operational requirements. With typical active power consumption around 1 W, it remains within the 10 W power budget and supports the high sampling rates necessary to achieve sub-200 ms latency.

Despite these advantages, the BGT60ATR24C lacks an AiP, requiring external patch antennas fabricated on high frequency, low loss substrates. These antennas must be precisely dimensioned and tuned to achieve the desired beam pattern and gain, which significantly increases RF design complexity and cost

compared with modules that integrate antennas. Errors in substrate selection, trace impedance, or antenna layout can degrade system sensitivity, detection range, and angular resolution. Additionally, because the device outputs only raw IF signals over an SPI interface and does not include on chip digital signal processing, all radar processing (including range FFTs), doppler extraction, micro motion isolation, and classification, must be implemented on an external processor or dedicated DSP. This requirement places considerable computational demand on the host microcontroller, such as the ESP32, and may necessitate additional processing resources to maintain real time performance.

Infineon provides basic driver support and evaluation tools for the BGT60ATR24C but does not supply application-level software or prebuilt algorithms for our stretch goal for vital sign detection. Consequently, respiration and heartbeat monitoring would require developing the entire signal processing and classification pipeline from the ground up, which would increase development time and technical risk. While the BGT60ATR24C's additional channels and potential for enhanced angular resolution offer clear advantages over the BGT60TR13C, these benefits are offset by more complex RF design requirements, the need for specialized materials, and substantial firmware development. Although the device could theoretically meet the range and latency requirements, including detection through drywall, achieving this level of performance would require significant engineering effort and would likely extend the project timeline beyond the scope of a prototype stage student project.

3.4.3.3. TI IWR6843AOP EVM

The Texas Instruments IWR6843AOP evaluation module is a fully integrated 60–64 GHz FMCW radar solution designed to accelerate the development of advanced sensing applications such as vital-sign detection and presence monitoring. It incorporates three transmit and four receive channels with a factory-calibrated antenna-on-package (AoP), eliminating the need for external antenna design and significantly reducing RF layout complexity. In addition to the radar front end, the module integrates a C674x digital signal processor (DSP) and an Arm Cortex-R4F microcontroller core, enabling on board signal processing that includes range FFTs, doppler estimation, and micro doppler extraction in real time. This on chip processing capability allows the radar to deliver processed data directly over UART or USB interfaces, reducing the computational load on the host microcontroller and simplifying system-level integration.

The IWR6843AOP EVM supports end-to-end latencies of under 200 ms, which meets the project's real-time performance requirements. It operates within the unlicensed 60 GHz ISM band under FCC Part 15 and is rated for industrial temperature ranges from −40 °C to +85 °C. With an active power consumption of approximately 2.0–2.5 W, it fits comfortably within the system's 10 W power budget, and its 9 x 6 cm footprint is acceptable to incorporate into the overall design. Moreover, Texas Instruments provides an extensive software ecosystem for this platform, including the mmWave SDK and mmWave Studio GUI, which streamline configuration, data acquisition, and algorithm development. Pre-

validated reference labs for vital sign and occupancy detection allow rapid prototyping without requiring the development of signal processing algorithms from scratch. TI has also demonstrated respiration detection through drywall at distances of 2 m or more, validating the module's suitability for stretch objectives of this project. Although its cost is higher than that of discrete radar ICs, the IWR6843AOP's high level of integration, mature software support, and proven performance substantially reduce development time and technical risk, making it a strong candidate for rapid deployment in a prototype system.

3.4.3.4. TI AWR1843BOOST EVM

The Texas Instruments AWR1843BOOST is a closely related development platform that offers many of the same hardware and software advantages as the IWR6843AOP EVM. It integrates three transmit and four receive channels along with an on-chip C674x DSP and Arm Cortex-R4F core, enabling real time signal processing, range doppler estimation, and micro motion analysis directly on the device. The AWR1843BOOST supports similar end-to-end latency and leverages the same mmWave SDK and mmWave Studio GUI, providing equivalent software support and development workflows. With an active power consumption of approximately 2.5 W and a footprint of about 10 x 6 cm, it also aligns with the goals and objectives of this project.

The primary difference between the AWR1843BOOST and the IWR6843AOP lies in its operating frequency band. The AWR1843BOOST operates in the 76–81 GHz automotive radar band, which offers slightly higher range resolution due to the shorter wavelength but is subject to stricter regulatory constraints and licensing requirements. This frequency range is optimized for automotive sensing applications, such as adaptive cruise control and collision avoidance, rather than detecting human presence. As a result, while the AWR1843BOOST delivers similar performance and processing capabilities, its operation outside the unlicensed 60 GHz ISM band makes it less aligned with the objectives of this project. Additionally, its higher cost compared with the IWR6843AOP further reduces its suitability for a cost-sensitive, portable prototype. Overall, although technically capable, the AWR1843BOOST is less optimal for the intended use case due to regulatory limitations and application focus.

3.4.3.5. Waveshare HMMD 24 GHz mmWave Radar

The Waveshare HMMD is a compact 24 GHz continuous-wave Doppler mmWave radar module providing presence detection, distance estimation, and vital-sign extraction over a UART interface. The module outputs a structured 35-byte frame at approximately 10 Hz containing a presence result byte, a raw distance value in centimetres, and 16 gate energyvalues at 70 cm spacing covering a detection range of up to approximately 11 m. Operating at 24 GHz, the free-space wavelength is approximately 12.5 mm - roughly five times longer than the primary IWR6843AOP's 60–64 GHz wavelength - giving the module substantially lower attenuation when penetrating non-metallic building materials suchas drywall, wood framing, and glass. This makes 24 GHz well suited for through-wall vital-sign detection in scenarios where the 60 GHz primary

radar's shorter wavelength would suffer excessive wall attenuation. In the 2HADRAD system, the Waveshare HMMD is integrated into the stationary anchor unit on an Arduino Uno R4 WiFi, operating independently over UART (Serial1, D0/D1). The anchor unit runs a custom DSP pipeline that performs log-compressed gate energy buffering, Hann-windowed FFT-based breathing rate extraction across three age bands (Adult, Child, and Infant), autocorrelation-based cardiac rate extraction with breathing harmonic notching, and Kalman-smoothed output. The anchor's fixed mounting position eliminates motion-induced phase noise that would corrupt micro-Doppler breathing extraction on the handheld primary device, making it particularly effective for through-wall sensing where the primary device may not maintain a stable pointing angle. Breathing rate, heart rate, and distance estimates from the anchor are transmitted to the main ESP32 over BLE and fused with primary IWR6843AOP readings. The module operates at 3.3–5 V, draws under 500 mW active, and requires no external antenna or RF layout design, making integration straightforward. Its low cost ($10.99) and UART-based interface make it directly compatible with the Arduino Uno R4 WiFi without additional hardware.

Radar Module	Infineon BGT60TR13C	Infineon BGT60ATR24C	TI IWR6843AOP EVM	TI AWR1843BOOST	Waveshare HMMD
Frequency (GHz)	60	60	60–64	76–81	24
Channels (Tx/Rx)	3TX/1RX	4TX/2RX	4TX/3RX	4TX/3RX	1TX/1RX
On-Chip Processing	No	No	C674x DSP + ARM R4F	C674x DSP + ARM R4F	Presence and distance output, no DSP core
Power (W)	0.5–0.6	1	2.0–2.5	2.5	≤0.5
Range Capability	≈ 2 m (requires DSP)	≥ 3 m (external antennas)	≥ 2 m through drywall	≥ 2 m (automotive band)	≥ 6 m LOS, through drywall and wood
Size (mm)	6.5 × 5.0	8.0 × 8.0	90 × 60	100 × 60	35 x 35
SDK Support	Drivers only	Drivers only	Full mmWave SDK + GUI	Full mmWave SDK + GUI	UART, open byte format
Approx. Price (USD)	$25 (chip only)	$45 (chip only)	$299 (EVM board)	$349 (EVM board)	$10.99

Table 12: Radar Module Comparison Table

3.4.3.6. FMCW Principle and Range-Doppler Processing

FMCW radar is a widely used technique for range and velocity sensing in automotive, industrial, and biomedical applications due to its high accuracy, continuous operation, and its relatively low power consumption. Unlike pulsed radar, which transmits data using discrete bursts of energy, FMCW radar continuously transmits a frequency modulated signal whose instantaneous frequency varies linearly with time, forming what is known as a chirp. The transmitted signal is defined as:

$$s(t) = A_t cos\left[2\pi \left(f_0 t + \frac{B}{2T_c} t^2 \right) \right]$$

where A_t is the transmit amplitude, f_0 is the start frequency, B is the total sweep bandwidth, and T_c is the chirp duration.

As the signal circulates through space and reflects off a target at range R, it experiences a round-trip time delay defined as:

$$\tau = \frac{2R}{c}$$

where c is the speed of light. When the reflected signals are mixed with a copy of the transmitted signal, the result is an intermediate frequency (IF) signal whose frequency is proportional to the time delay, and consequently to the target range. This beat frequency is expressed as:

$$f_b = \frac{B\tau}{T_c} = \frac{2BR}{cT_c}$$

From this relationship, we can derive the range R to the target as:

$$R = \frac{cT_c}{2B} f_b$$

In addition, the range resolution of the radar system is determined by the total sweep bandwidth and is given by:

$$\Delta R = \frac{c}{2B}$$

A larger sweep bandwidth enables a finer discrimination between closely spaced targets. To process the received signal, the radar samples the IF signal and applies the Fast Fourier Transform (FFT) across each chirp. This transforms the time domain signal into the frequency domain, allowing the system to resolve multiple targets as discrete range bins, each corresponding to a unique beat frequency. A secondary FFT, known as the Doppler FFT, is administered to extract velocity across multiple chirps to observe how the signal phase evolves. Together, this signal processing approach enables FMCW radar to detect, localize, and track multiple static and dynamic targets simultaneously with high precision. All four radar modules considered implement this FMCW principle, but those without on chip DSP (such as the Infineon devices) require the host MCU to perform these FFT and doppler operations externally, whereas the TI modules execute them internally and deliver processed data directly.

3.4.3.7. Micro-Doppler Extraction for Vital signs

Vital sign detection depends on the ability to sense millimeter-scale chest displacements caused by respiration and heartbeat. These minute motions create periodic phase modulations in the returned signal that appear as low frequency oscillations superimposed on the static reflection. The DSP isolates this micro doppler component by applying high pass filtering to remove clutter and low pass filtering to capture the breathing and heartbeat bands (typically 0.1–0.5 Hz for respiration and 1–2 Hz for cardiac motion). The resulting waveform is processed through time domain analysis or FFT based spectral estimation to output real-time breathing and pulse rates. In devices like the BGT60TR13C and BGT60ATR24C, these processing tasks must be implemented manually on the external MCU, whereas the IWR6843AOP and AWR1843BOOST handle micro doppler extraction internally, simplifying integration and reducing software complexity.

3.4.3.8. On Chip Processing and Firmware Framework

The TI IWR6843AOP radar integrates a C674x DSP and an ARM Cortex-R4F microcontroller core to handle signal processing and system control in parallel. The DSP executes computationally intensive tasks such as FFTs, filtering, and peak detection, while the ARM core manages radar configuration, data framing, and communication with the external ESP32 microcontroller via UART. This dual-core architecture allows one processor to continuously compute the next radar frame while the other handles data output, maintaining end-to-end latency below 200 ms. The TI mmWave SDK provides a layered software framework that abstracts low-level hardware control from higher-level application logic. Its components include the mmWave link driver for device-level commands, a control layer for interpreting configuration messages, and an application layer for processing and communication. Pre-validated reference labs within the SDK accelerate development of occupancy and vital-sign monitoring applications, while the mmWave Studio GUI enables rapid configuration, data capture, and visualization. Together, these tools reduce the need for custom firmware development and shorten the path from prototype to functional system.

3.4.3.9. Chirp Design, Data Handling, and Integration

Radar performance depends heavily on chirp design parameters such as slope, bandwidth, and repetition period, which together determine range resolution and frame rate. Increasing bandwidth improves range resolution $\Delta R = \frac{c}{2B}$ but at the cost of computational load. Through experimental calibration, chirp parameters will be tuned to maintain sub 200 ms update intervals while potentially detecting vital signs at distances ≥ 2 m through drywall.

Processed radar data are streamed as Type–Length–Value (TLV) packets over a UART interface. Each TLV block contains metadata such as frame number, object count, range–Doppler matrices, and micro-Doppler waveforms, which the system uses to extract and display motion parameters. Integration between the radar module and the ESP32-WROOM microcontroller forms the backbone of the embedded data pipeline. The ESP32 was selected for its dual-core architecture, low power consumption, and robust support for UART, SPI, I2C, Wi-Fi, and

Bluetooth, enabling seamless communication between the sensing front end and the control or cloud layers.

A primary UART channel operating at 921,600 baud is used to transmit radar data from the IWR6843AOP to the ESP32. One core of the ESP32 is dedicated to UART reception and preprocessing, while the second handles higher level tasks such as wireless communication or UI updates. A secondary UART or SPI interface is reserved for radar configuration, firmware updates, and diagnostic output. SPI offers higher bandwidth for future scalability, but UART is prioritized in this prototype for its simplicity, native support within the ESP-IDF development framework, and sufficient throughput for the system's expected data rates.

Precise synchronization between the radar and MCU is achieved using interrupt-driven GPIO signaling. A "frame ready" pulse from the radar notifies the ESP32 to initiate a data read, ensuring low-latency data handling and consistent timing alignment across measurement cycles. Additional GPIO status lines provide feedback on radar states such as chirp activity or fault detection, allowing the MCU to respond immediately without continuous polling. The system also employs controlled power sequencing to ensure stable startup: the 3.3 V logic rail is enabled first, followed by the analog front end. A dedicated reset line controlled by the ESP32 reinitializes the radar in case of communication timeouts or thermal faults, further improving system robustness. These integration strategies are significantly simplified by TI's development kits, which provide defined communication protocols and structured data formats, whereas integrating Infineon's devices would require custom data framing and additional software layers to manage raw IF data streams.

3.4.3.10. Mechanical, EMI, and Thermal Design Constraints

Operating at 60 GHz imposes stringent electromagnetic and mechanical design requirements. High frequency traces can radiate or couple noise into digital lines, degrading performance. To mitigate this, the radar module will be isolated from digital circuits and enclosed in a grounded metal shield, which also provides mechanical stability and environmental protection. Ground planes beneath the radar are continuous and connected through fences to minimize impedance discontinuities. Analog, digital, and power domains are carefully separated to reduce cross coupling. The radar dissipates approximately 2 W of power during operation, which is managed using thermal vias or small plated through holes that conduct heat into internal copper layers and the enclosure. The chassis will be constructed from aluminum or copper-plated polymer for efficient heat dissipation while remaining lightweight. The system is designed to operate reliably between −40 °C and +85 °C, exceeding the 0–50 °C requirement for the application. Validation testing will include temperature cycling and vibration tests to ensure vigorous performance in challenging environments. While all candidate radars share similar thermal and EMI considerations, the integrated design of the IWR6843AOP reduces RF layout complexity and minimizes the risk of impedance mismatches, which would be more challenging to control when implementing external antennas for the BGT60ATR24C.

3.4.3.11. Radar Selection Justification

After evaluating candidate radar devices, we compared the tradeoffs between integrating a bare mmWave radar IC and using a fully integrated development kit. Bare ICs such as Infineon's BGT60TR13C and BGT60ATR24C offer low unit cost and compact size, but they shift critical design burdens onto the development team, including RF front end design, antenna tuning, and signal processing implementation. These tasks require high frequency PCB layout on specialized substrates with controlled impedance and strict trace tolerances, where even small design errors can severely degrade performance. They also require development of custom signal processing algorithms for vital signs and presence detection from the ground up. In comparison, development kits like TI's IWR6843AOP and AWR1843BOOST integrate these elements into a single platform, which not only eliminates most of the RF and antenna challenges but also offloads significant signal-processing work from the host MCU. The TI mmWave SDK, mmWave Studio GUI, and validated reference labs provide ready to use tools for configuring chirps, visualizing data, and building classification algorithms, enabling rapid system level progress. Unlike Infineon's options, which provide only low-level drivers and require building the processing pipeline from scratch, TI's ecosystem shortens development time by providing pre-validated application code and visualization utilities. Although the development kit has a higher cost, it eliminates hidden costs such as custom PCB fabrication, RF test equipment, and extended debugging time. For a student project with tight timelines, the overall cost to benefit ratio strongly favors the development kit. Furthermore, once a proof of concept is validated, firmware and algorithms developed on the kit can be directly ported to a custom board using the same IC, lowering cost in production.

After evaluating multiple radar options, the Texas Instruments IWR6843AOP EVM was selected as the radar front end for our system. It uniquely combines AoP integration, on chip DSP, and a mature software ecosystem at 60 GHz, features that collectively minimize RF design complexity and accelerate development. Its proven capability to detect vital signs through drywall at distances of 2 m or more aligns directly with our stretched application requirements. Compared with Infineon's BGT60TR13C and BGT60ATR24C, which both require external antennas, off chip signal processing, and extensive firmware development, the IWR6843AOP provides a plug and play platform that integrates these elements into a single, validated solution. Although the Infineon devices offer lower unit cost, their reliance on external DSPs and complex PCB design would significantly increase development time, require specialized fabrication materials, and raise the risk of performance degradation due to RF layout errors. Likewise, while the AWR1843BOOST offers similar hardware integration, its operation in the licensed 76–81 GHz band introduces regulatory challenges and is optimized for automotive radar rather than sensing human presence, making it less aligned with the system's intended application.

The IWR6843AOP EVM enables us to achieve first measurements immediately, focus engineering effort on system level integration and maintain a clear

migration path to a custom board for future production. Furthermore, it’s built in SDK, structured data output, and well documented communication protocols significantly simplify integration with the ESP32 microcontroller, a level of software support not available for the Infineon alternatives. In terms of long-term scalability, firmware and algorithms developed on the IWR6843AOP can be directly ported to a custom board using the same IC, ensuring a smooth transition from prototype to production without re-architecting the signal processing pipeline. This combination of performance, cost efficiency, regulatory suitability, and ecosystem support makes the IWR6843AOP the optimal radar choice for our prototype and the most practical solution to meet the system’s sensing, latency, and integration requirements.

The Waveshare HMMD 24 GHz mmWave radar was selected as a second independent sensing node integrated into the stationary anchor unit. While the IWR6843AOP was chosen for its high-resolution 60–64 GHz FMCW vital-sign extraction on the primary handheld device, the Waveshare HMMD complements it by providing through-wall detection capability from a fixed vantage point. Its 24 GHz operating frequency produces a free-space wavelength approximately five times longer than the primary radar, resulting in substantially lower attenuation through non-metallic construction materials such as drywall and wood framing. This makes it well suited for detecting subjects in adjacent spaces where the primary device cannot achieve line-of-sight. The module's fixed mounting position on the anchor eliminates the motion-induced phase noise that would corrupt micro-Doppler breathing extraction on a handheld platform, improving vital-sign signal quality from the stationary vantage point. At $10.99 with a simple UART frame protocol requiring no external SDK or DSP configuration, the Waveshare HMMD offers a low-cost, low-integration-overhead path to independent through-wall physiological sensing that meaningfully extends the detection coverage of the overall 2HADRAD system beyond what the primary radar alone can provide.

3.4.4. IR Sensor

3.4.4.1. AMG8833 IR Thermal Camera (Grid EYE)

The Adafruit AMG8833 is an 8×8-pixel IR thermal sensor array that detects emitted heat energy and converts it into temperature data transmitted over an I2C interface. Operating at 3.3 V logic and drawing less than 10 mA, it is fully compatible with the ESP32WROOM microcontroller and ideal for low power embedded applications. This component will measure temperatures ranging from 0°C to 80°C (32°F to 176°F) with an accuracy of +- 2.5°C (4.5°F). It can detect a human from up to 7 meters (23) feet and has a maximum frame rate of 10Hz. In this project, the AMG8833 is programmed to identify temperatures within the typical human body range (30 °C - 40 °C), triggering a “Human Detected” message on the LCD display when such readings are observed. Its straightforward I2C communication, compact design and existing Arduino and MicroPython library support make it a highly practical choice for integration. This unit costs about $50 making it an affordable option. While it can produce low

resolution thermal maps as a part of our stretch goals, its primary advantage lies in its fast and reliable detection of heat signatures for life verification in conjunction with the other radar subsystems.

3.4.4.2. MLX90640 32×24 Thermal Camera Array

The MLX90640 offers a higher resolution 32x24 pixel thermal grid, producing more detailed thermal information than the AMG8833. It also operates over I2C and runs at 3.3 V logic, compatible with the ESP32WROOM. The module supports frame rates from 0.5 Hz to 16 Hz and provides a detection range of roughly 1 m - 7 m with a field of view between 55° and 110°, depending on the lens version. Typical power consumption ranges from 18–23 mA, making it moderately efficient but still suitable for battery powered applications. The additional resolution would allow more refined temperature mapping or a smoother heat map display as a part of our stretch goal. However, this improvement comes with higher computational demands and a cost around $60, which is more than the AMG8833 but still falls within the project budget. For a system prioritizing fast and reliable presence confirmation over detailed imaging, the MLX90640 may exceed the project's immediate needs, though it remains an excellent candidate.

3.4.4.3. MLX90641 16×12 Thermal Array

The MLX90641 is a midrange alternative that balances resolution and cost between the AMG8833 and the MLX90640. It provides a 16×12 thermal grid, uses the same I2C communication protocol, and operates at 3.3 V logic. With a typical field of view of 55° and a detection range of up to 7 m, it offers better spatial accuracy than the AMG8833 while remaining easier to process than the MLX90640. Power consumption is moderate at around 25 mA, and the sensor includes built-in calibration for temperature stability. For this project, it could enhance detection confidence by mapping larger areas, but the smaller pixel count compared to the MLX90640 limits fine detail for visualization tasks. Its moderate price point of $40 and simpler data handling makes it a strong backup option.

3.4.4.4. FLIR Lepton 3.5 Thermal Imaging Module

The FLIR Lepton 3.5 is a professional grade micro thermal camera core offering 160×120 resolution and radiometric output. It interfaces via SPI through a dedicated breakout board, such as the PureThermal 2, and supports 3.3 V logic. Although the FLIR Lepton 3.5 delivers excellent image quality and can operate through light smoke, it draws a relatively high current of about 150 - 200 mA, which makes it less practical for battery powered or portable systems where energy efficiency is a concern. The Lepton provides the most detailed heat data among the compared units and can detect temperature variations from several meters away with precision under +-2 °C. While this would allow full image thermal visualization on the LCD, integration complexity and price are significantly higher, exceeding $150 - $200 once a carrier board is included. Due

to its processing requirements and higher cost, the Lepton is better suited for advanced prototypes where high-definition thermal imaging is necessary.

3.4.4.5. Infrared Temperature Sensor (MLX90614)

The Grove MLX90614 is a single point, contactless IR thermometer capable of measuring surface temperatures from –70 °C to 380 °C. It operates at 3.3 V - 5 V and communicates via I2C. Power consumption is minimal at roughly 1.5 - 2 mA, making it highly efficient for low power systems. Rather than providing a pixel array, it outputs an average temperature reading of the target in its 35° field of view. While limited to one measurement point, its simplicity, low power usage, and $20 cost make it useful for basic proximity or presence detection based on a single temperature threshold. In this project, it could serve as a compact alternative for early testing but lacks the spatial awareness needed for multi-point detection or mapping.

3.4.4.6. Limitations

The limitation of IR thermal sensors is that they cannot detect heat sources through solid materials such as walls, doors, or glass. This is because IR radiation is absorbed or reflected by these surfaces, preventing the sensor from capturing the temperature of objects behind them. As a result, the IR module is limited to the line-of-sight detection, sensing only the surface temperature of whatever is directly visible within its field of view. To overcome this limitation, our team is exploring the integration of UWB radar and similar radio frequency sensing technologies, which can detect motion and presence through most nonmetallic barriers. By combining radar-based presence detection with the IR sensor's precise thermal verification, the system can achieve a more reliable form of life detection. The radar subsystem would provide distance or motion data even when visibility is obstructed, while the IR module would confirm whether the detected object falls within a human temperature range. The complementary sensing strategy enhances detection accuracy, minimizes false positives, and improves performance across a range of environmental and operational conditions.

Component	AMG8833(GRID-eye)	Melexis MLX90641	Waveshare MLX90640	FLIP Lepton 3.5	Grove MLX90614 (single point)
Array / Resolution	8x8 (64 px)	16x12	32x24	160x120	1 px (spot)
FOV (in degrees)	60 x 60	About 55	55 std or 110 wide	About 57x44	About 35
Interface	I2C	I2C	I2C	SPI	I2C
Interface	SPI	SPI	SPI +I2C	SPI	SPI
Voltage / Logic Compatibility	3.3 V	3.3 V	3.3 V	3.3 V	3.3 V

Component	AMG8833(GRID-eye)	Melexis MLX90641	Waveshare MLX90640	FLIP Lepton 3.5	Grove MLX90614 (single point)
Current (typ.)	10 mA	25 mA	18-23 mA	150-200mA	1.5-2mA
Outdoor Readability	Fair	Good	Very good	Very good	Fair
Pros	Easy wiring, very low power, good libraries	More pixels, light compute	Best balance of detail vs ease	Pro-grade images	Ultra simple, ultra-low power, cheap
Cons	Low resolution, best for indoor use or shade only	Lower resolution than 90640; costs more than AMG	Heavier processing than AMG, costs more	Expensive, more code/power	No spatial info, not ideal for human detection alone

Table 13: IR Thermal Sensor Comparison Table

3.4.4.7. Selected IR Sensor Justification

All evaluated IR modules are compatible with the ESP32WROOM through I2C communication and operate safely at 3.3 V logic. The main tradeoffs among them involve resolution, range, outdoor reliability, processing demand, and cost. The Waveshare MLX90640 provides a strong balance between performance and practicality, offering a significantly higher 32x24 pixel resolution compared to lower end arrays like the AMG8833 while maintaining straightforward integration and low power requirements. Its wider field of view and built in ambient temperature compensation make it more stable and accurate in outdoor or highlight environments, where simpler sensors can become unreliable. Higher end options like the FLIR Lepton enable professional thermal imaging but require more complex drivers and significantly more power. The Grove MLX90614 remains the simplest and least expensive choice but lacks the spatial awareness necessary for reliable human detection.

For this prototype, the Waveshare MLX90640 was selected because it offers a strong combination of resolution, outdoor reliability, and ease of integration. It communicates over I2C, operates at 3.3 V logic, and draws only about 20 mA, making it directly compatible with the ESP32WROOM without requiring additional hardware. The MLX90640's 32×24 thermal array allows for clearer detection patterns, improving accuracy in identifying human heat signatures even under bright sunlight or varying temperatures. Its built-in calibration and available Arduino and MicroPython libraries make implementation straightforward, enabling fast setup and minimal code modification. The sensor can also support our stretch goals such as displaying low resolution heat maps on the LCD or logging temperature distributions over time. Overall, it provides a significant upgrade in capability over the AMG8833 while maintaining simplicity, efficiency, and full compatibility with the current system design.

3.5. Safety and RF Exposure Management

In the development of our handheld emergency detection device, which utilizes the TI IWR6843AOP EVM for reliable human presence and potential vital sign detection during critical situations, incorporating a secondary motion detector is crucial for regulatory compliance and user safety. The radar's high frequency RF emissions, while certified under FCC Part 15 for the EVM as an intentional radiator, necessitate precautions to limit proximity exposure, aligning with FCC's Maximum Permissible Exposure (MPE) limits and Specific Absorption Rate (SAR) guidelines to prevent potential health risks from prolonged or unintended near-field contact. A proximity detector senses the presence of any object or person within a close range of 22–30 cm by measuring ultrasonic echoes, enabling a simple and reliable safety mechanism. This temporarily suspends transmission of the radar module whenever someone is detected nearby, ensuring compliance with FCC RF exposure limits during handheld use. The detector operates only when needed, draws minimal power, and maintains the device’s battery life and portability for unpredictable emergency situations. Since the UWB module is already fully compliant, this proximity system is used exclusively as a safety interlock for the radar, providing an extra layer of protection for the operator and anyone nearby without affecting the device’s core human-detection functionality.

3.6. Safety and RF Exposure Management Part Selection

3.6.1. HC-SR04 Ultra Sonic Sensor (USS)

For our handheld emergency detection device utilizing the TI IWR6843AOP EVM radar module, the HC-SR04 ultrasonic sensor serves as the baseline for proximity-based motion detection to enforce a 22-30 cm safety buffer, deactivating RF emissions in compliance with FCC regulations. This device utilizes sound waves to measure the time it takes for an echo to return, providing reliable short-range detection of any solid object or movement in front. It has a detection range of 2 cm to 400 cm and a narrow 15° beam angle, making it suitable for precise frontal monitoring in cluttered rescue environments. The device’s simple digital PWM output integrates seamlessly with common microcontrollers (MCUs), requiring minimal power (an average of 15 mA) and only basic trigger-echo pin setup for calibration. This makes it ideal for battery-powered portability. However, it may be susceptible to acoustic interference from wind or soft surfaces, although this is mitigated in our controlled handheld use case.

HC-SR501 Passive Infrared Sensor

Another excellent alternative is the HC-SR501 Passive Infrared (PIR) sensor, which detects motion through changes in IR radiation from warm bodies like humans, providing a broader detection envelope up to 7 meters with a 100° conical field of view, far exceeding the HC-SR04's precision focus for enhanced front-facing awareness in dynamic emergency scenarios. Unlike the ultrasonic

approach, the PIR excels at human specific motion without distance measurement, triggering instantly on heat signatures to pause the radar module preemptively, and its adjustable sensitivity and delay (0.5-200 seconds) allow customization to align with our 22-30 cm threshold when fused with the USS data. Operating on 5-20V with ultra-low quiescent current, it draws negligible power for standby modes, connects via simple digital GPIO, and costs under $2, rendering it a cost-effective, low-profile addition that avoids false triggers from non-living objects while complementing the HC-SR04's proximity accuracy.

3.6.2. VL53L0X Time of Flight (ToF) Laser Distance Sensor

Another strong contender is the VL53L0X ToF laser distance sensor. It uses invisible IR laser pulses to calculate distances based on the time it takes for the light to travel. This sensor offers sub-millimeter accuracy up to 2 meters and has a compact 25° beam, making it suitable for bridging the gap between the HC-SR04's short-range utility and the broader motion need of our device. This optical method provides high-resolution motion profiling by tracking rapid distance fluctuations indicative of human approaches, enabling algorithmic deactivation of the radar at the exact 22-30 cm mark with 50 Hz update rates to outpace ultrasonic echoes in speed and precision, even in low light or dusty conditions common to emergencies. At $5-10, with I2C interface for easy MCU integration and low 20 mA active draw, the VL53L0X's tiny 4.4 x 2.4 mm footprint fits seamlessly into the handheld enclosure, though it may require shielding from direct sunlight; overall, it enhances reliability over the HC-SR04 by reducing environmental sensitivities.

Feature	HC-SR04	HC-SR501	VL53L0X
Detection Type	Ultrasonic echo	Infrared heat changes	Laser ToF
Range	2 cm – 400 cm	Up to 7 m	Up to 2m
FoV	15º narrow beam	100º cone	25º narrow beam
Accuracy	+/- 3 mm	Medium ~1 m	High +/- 3%
Power Consumption	15 mA	<50 µA	20 mA
Interface	Digital PWM	Digital GPIO	I2C
Size	45x20x15 mm	32x24 mm	4.4x2.4 mm
Pros	Precise short-range proximity	Broad human motion detection	High precision approach tracking
Cons	Sensitive to environmental changes	No distance information	Ambient light interface

Table 14: Safety and RF Exposure Management parts comparison

3.6.3. Safety and RF Exposure Management Part Selection Justification

For the proximity-based safety interlock required to suspend transmission of the TI IWR6843AOP radar module when any object or person enters the 22–30 cm safety zone, the HC-SR04 ultrasonic sensor was selected as the primary solution. This choice leverages an already-proven, in-hand component that

provides direct, accurate distance measurement (with ±3 mm resolution) rather than simple motion triggering, allowing the system to enforce a precise and repeatable deactivation threshold, critical for consistent FCC RF exposure compliance. In addition to enforcing the safety buffer, the ultrasonic measurements support continuous environmental scanning as the operator moves the handheld unit through a space. The HC-SR04 operates entirely without RF emissions, consumes only ~15 mA during active pings (negligible when duty-cycled), interfaces directly with most common MCUs via two GPIO pins, and requires no additional calibration or shielding in indoor/outdoor emergency environments. When an object is detected within a configurable alert range (~2m), the system prompts the operator to confirm the target using higher-fidelity sensors such as mmWave radar or thermal imaging, reducing false positives while prioritizing areas of interest. While the HC-SR501 PIR sensor could be explored in future revisions for broader motion coverage or as a redundant trigger, its lack of inherent ranging capability makes it less suitable for the strict, distance safety buffer required in the current project scope. The HC-SR04 therefore represents the simplest, most reliable, and immediately implementable solution that fully satisfies the regulatory and functional requirements.

3.7. User Interface

The UI provides the operator with tactile, visual, and auditory interaction with the system. The device is designed for search and rescue environments, so its interface must remain fully functional even in challenging conditions like low visibility, operation through gloves, and high ambient noise. To meet these demands, the UI incorporates five essential components: pushbuttons, an LED, a speaker, and an LCD display module. The pushbuttons serve as primary input controls, allowing operators to initiate scans, toggle modes, or reset the system with ease. The LED provides immediate visual and auditory feedback for system states and alerts, while the speaker supports more detailed audio cues. The LCD display module offers clear, structured data visualization, presenting vital information such as sensor readings, system status, and detected targets in real time. Collectively, these components create a redundant and multimodal interface that guarantees reliable user feedback and control, even when one sensory channel, such as sight or hearing, is compromised by environmental factors.

3.7.1. Pushbutton

Pushbuttons were selected as the primary input mechanism due to their mechanical simplicity, high reliability, and superior tactile feedback in demanding field conditions. Unlike capacitive or touch-sensitive controls, which can become unresponsive when operated through gloves or contaminated surfaces, mechanical pushbuttons provide a distinct physical click that confirms successful actuation. This tactile confirmation is critical in search and rescue scenarios where first responders often wear heavy protective gloves, turnout gear, or bulky

sleeves that reduce fine motor control. The ability to engage controls without precise finger positioning or visual confirmation is essential for usability and safety in these environments. Their straightforward electrical interface, based on momentary contact closure, also makes pushbuttons easy to debounce and integrate into low power embedded systems. Additionally, sealed pushbuttons offer excellent resistance to dust, moisture, and vibration, making them well suited to the harsh environmental conditions encountered during field deployment. While membrane keypads or capacitive sensors might reduce moving parts, they would perform poorly under the thick gloves and limited dexterity common among firefighters and rescue personnel, and they are more prone to false triggers from debris or moisture.

3.7.2. LED Indicator

LED indicators were chosen as the primary feedback mechanism for this system because they provide instant response, low power consumption, and strong reliability in demanding field environments. They give immediate visual confirmation of key system states scanning (red held on) and detection (red flashing), which is especially important when audible alerts might be drowned out by ambient noise or when tactile feedback alone is not enough. For first responders working in smoke-filled, dark environments who often while wearing helmets, face shields, and respirators, maintaining visibility is a serious design requirement. LEDs help meet that challenge by producing bright, directional light with wide viewing angles, allowing status information to remain visible even when the device is not positioned directly in front of the user or is viewed through layers of protective gear.

Another reason LEDs are ideal for this application is that they simplify the overall system design. Surface mount device (SMD) LEDs reduce panel wiring, streamline PCB layout, and maintain mechanical durability under shock and vibration, all of which are crucial for a portable device that might be dropped, jostled, or used in extreme conditions. While alternatives like LCD or OLED panels could display more detailed information, they consume more power, are harder to see through masks and face shields, and are generally less rugged in harsh search and rescue conditions.

Brightness is also a key consideration. Luminous intensity in the 20–80 mcd range is usually sufficient for handheld devices at close range, but higher output becomes necessary in smoke, bright sunlight, or other high-contrast environments. Wide viewing angles further improve peripheral visibility and reduce the need to precisely orient the device, a valuable feature when responders are focused on victims rather than equipment. The LED families under consideration provide a range of luminous intensities and maintain wide viewing angles, allowing us to tailor individual indicators for continuous status signaling or high priority alerts.

Finally, package size plays an important role in balancing performance with physical design constraints. Most of the LEDs being evaluated are available in standard 0603 (1.6 × 0.8 mm) and 0805 (2.0 × 1.25 mm) SMD packages. 0603

LEDs are excellent for space-constrained layouts and automated assembly, while 0805 LEDs offer greater luminous output and improved mechanical strength, which is especially useful in ruggedized systems. Choosing components from families that offer both sizes gives us flexibility to trade off brightness, board space, and mechanical durability without sacrificing electrical compatibility or manufacturing efficiency.

3.7.3. Buzzer

A buzzer was considered as a part of the UI to provide reliable auditory feedback in conditions where visual or tactile cues may be missed. Sound is a critical alert channel in low visibility environments in smoke-filled rooms, where first responders are wearing face shields, respirators, and heavy gear that limit their field of view. Unlike visual indicators, sound can penetrate obstacles and immediately attract attention, even when the device is not in direct line of sight. By generating distinct tone patterns, such as short chirps for power on, continuous tones for life detection, or rapid bursts for fault conditions, the system can convey status information automatically without requiring the operator to read a display.

In addition to volume, tone control and sound profile flexibility are important considerations. A buzzer that can be modulated by the microcontroller allows different alerts to be assigned to different events, reducing cognitive load and improving reaction time. Other design factors include current consumption, physical footprint, and mounting compatibility with the device’s enclosure.

3.7.4. Speaker

A dedicated speaker is being considered in the UI to provide continuous or modulated audio feedback that communicates more complex information than a simple buzzer can. While the buzzer is ideal for quick alerts and system status notifications, the speaker enables real time auditory representation of radar derived signals, such as breathing or heartbeat waveforms. This feature provides first responders with an intuitive sense of life presence even when they cannot look at a display. For example, while navigating debris, carrying equipment, or wearing full protective gear that limits visibility and skill. For this project application, the speaker must balance several requirements: clarity, efficiency, durability, and compact.

3.7.5. Display Screens

An LCD screen is a flat panel display technology that uses liquid crystals to control light and present information such as text, numbers or graphics. The liquid crystals twist or untwist when an electrical signal is applied, altering how much light can pass through. A backlight, or ambient light in some designs, provides illumination while polarizers and color filters shape the light into visible images. In essence, an LCD functions as a light gate controlled by electronics, enabling it to display clear and dynamic visual information.

Although there are several different display technologies available, we chose to focus on LCDs because they align well with our course history, prior experience and existing background knowledge. By leveraging a technology, we are already familiar with, we can efficiently develop a working product by ensuring reliable integration into our system.

For this project, the LCD serves as a critical component of the UI, providing detailed feedback that cannot be conveyed by an LED or the audio amplifier alone. At the basic level, the display must clearly present simple text outputs such as “Human Detected” “Hazard” or “Safe Zone” ensuring that the user receives unambiguous information in real time. As the system progresses to advanced goals, the LCD should be capable of showing multiple sensor readings simultaneously, supporting simple icons for hazards, and offering improved clarity for field use. For stretch goals, the display would need to handle more complex visualizations, such as logged event history, battery monitoring, or graphical overlays for sensor data trends. To meet these requirements, the LCD must be compatible with the ESP32WROOM microcontroller, supporting standard communication protocols such as SPI or I2C for seamless integration.

3.8. User Interface Parts Selection

3.8.1. Pushbutton

3.8.1.1. Apem 1ZW0913611806

The **APEM 1ZW0913611806** is a durable pushbutton switch designed for use in demanding outdoor and industrial environments. It offers a robust construction and can be paired with compatible switch assemblies that provide environmental sealing and LED illumination for improved visibility in low-light conditions. Its clearly marked actuation surface and tactile feedback make it easy to operate, even when users are wearing gloves or have limited dexterity. This makes the switch particularly useful in high-stress or low-visibility scenarios where quick and reliable activation is critical. However, the switch’s larger mounting footprint may increase panel space requirements and overall system weight, which could present challenges in compact or handheld devices. Additionally, while the larger actuation surface improves usability with thick gloves, it may be excessive for lightweight prototypes where multiple controls must fit within a small enclosure.

3.8.1.2. E-Switch PV6F24011

The **PV6F24011** is a vandal-resistant pushbutton switch designed for use in demanding outdoor and industrial environments. As part of the PV6 series, it features a durable metal construction, panel-mount design, and integrated LED illumination options that improve visibility in low-light conditions. Its round, flush actuator provides a large and clearly defined actuation surface, allowing users to operate the switch reliably even while wearing gloves or when dexterity is limited. This makes it particularly useful in high-stress or low-visibility environments where quick and dependable activation is essential. However, the switch’s 16 mm panel cutout and metal housing increase panel space requirements and overall system weight, which can be challenging in compact or handheld devices.

Additionally, while the larger actuator improves usability with gloves, it may be excessive for lightweight prototypes where multiple controls must fit within a small enclosure.

3.8.1.3. C&K PTS645VL58-2 LFS

The C&K PTS645VL58-2 LFS is a standard 6 x 6 mm low profile tactile switch that offers a crisp tactile feel and long mechanical life at low cost. Its compact size and simple mounting make it easy to incorporate into dense circuit board layouts. However, this switch lacks environmental sealing and is vulnerable to dust and moisture access, which significantly limits its suitability for field deployed devices. More importantly, its small size and low actuation force make it difficult to operate reliably with gloved hands or reduced dexterity, a critical requirement in first responder environments. While its low cost and compact footprint make it a useful option for early-stage prototyping or indoor applications, it does not meet the environmental or ergonomic requirements of a ruggedized search and rescue device.

Part Number	Type	Actuation Force	Footprint size	UI experience	Mount style
1ZW0913611806	5 way navigation	~ 2.5 N	Compact	Intuitive	PCB mount
C&K JS202011AQN 5-Way Nav Switch	5 way navigation	~ 2 N	Medium	Intuitive	Through hole PCB
PV6F24011	SPST momentary pushbutton	~ 5 N	Requires panel cutout	Requires multiple buttons	Panel mount

Table 15: Pushbutton Comparison Table

3.8.1.4. Pushbutton Selection Justification

The APEM 1ZW0913611806 was selected as the primary pushbutton interface for the system because it provides the most balanced combination of durability, usability, and compact integration for the intended application. The switch offers a clearly defined actuation surface and tactile feedback that allows reliable operation even when users are wearing gloves or operating the device under stressful conditions. Its compact form factor also allows it to be easily integrated into the enclosure without significantly increasing panel space requirements or system weight, making it well suited for a portable prototype.

Alternative switches were evaluated during the design process. The E-Switch PV6F24011 provides excellent durability and vandal-resistant construction, making it well suited for harsh industrial environments. However, its larger panel-mount design increases both enclosure size and weight, which can be impractical for compact handheld systems where space and portability are

important considerations. The C&K PTS645VL58-2 LFS offers a low-cost tactile switch solution with a small footprint, but its limited environmental protection, smaller actuation surface, and reduced usability with gloved hands make it less suitable for field conditions where reliability and ease of actuation are critical.

By providing reliable tactile feedback, a recognizable actuation interface, and a compact design that simplifies enclosure integration, the APEM 1ZW0913611806 offers the most practical solution for this prototype. Its balance of usability, durability, and space efficiency makes it well suited for a system that must remain reliable while operating in demanding environments.

3.8.2. LED

3.8.2.1. Kingbright KP-1608 Series (Green, Yellow, Red, Blue, Amber)

The Kingbright KP-1608 series is a family of 0603-package SMD LEDs known for their compact size, energy efficiency, and availability in multiple colors, including green (KP-1608CGCK), red (KP-1608SRCP), yellow (KP-1608SYCK), blue (KP-1608SURCK), and amber. The green variant offers approximately 30 mcd luminous intensity with a 120° viewing angle, providing excellent visibility in low light or indirect viewing conditions, while the red and yellow versions achieve similar performance at about 25–50 mcd and 25 mcd, respectively. Operating at a typical current of 2 mA, these LEDs are highly power efficient and appropriate for battery powered field devices. Their color diversity allows designers to standardize the footprint while customizing status indicators for optimal human factor performance. Compared with higher intensity 0805 LEDs, the KP-1608 series sacrifices some brightness but enables tighter PCB layouts and reduced power draw, making them a strong choice for continuous indicators such as “Power On” or “Scanning.”

3.8.2.2. Lite-On LTST-C191 Series (Red, Orange, Yellow, Green, Blue)

The Lite-On LTST-C191 series is another 0603-package LED family that emphasizes stable hue and reliable performance across a wide current range (2–20 mA). The yellow LTST-C191KSKT variant offers a luminous intensity of about 15 mcd, while the red and green versions provide around 20–30 mcd and 30 mcd, respectively. The series maintains consistent color performance under thermal stress, which is important in devices that may experience rapid temperature fluctuations in the field. Although these LEDs are slightly less bright than the Kingbright or OSRAM options, their thermal stability and color consistency make them suitable for indicators that must remain clearly visible during prolonged operation. The wide color range also enables flexible state assignment, which can be refined during field testing based on responder feedback.

3.8.2.3. OSRAM R976 Series (Red, Yellow, Orange, Blue)

The OSRAM R976 series is a family of high brightness 0805-package LEDs optimized for critical alerts and high visibility indicators. The LS R976 red LED

delivers up to 80 mcd luminous intensity with a 120° viewing angle, ensuring immediate visibility even through smoke, tinted visors, and peripheral vision. Yellow and orange variants offer luminous intensities of about 60–70 mcd and 60 mcd, respectively, while maintaining similar electrical characteristics. The slightly larger 0805 package increases mechanical robustness and provides enhanced heat dissipation, improving reliability under continuous use. Compared with 0603 options, the R976 series offers significantly higher brightness and enhanced visual impact, making it ideal for safety critical alerts such as “Detection Active” or “System Fault.” The tradeoffs include increased board space and marginally higher power consumption, but these are acceptable for high priority indicators that demand immediate attention.

Part Number	Manufacturer	Color	Package	Luminous Intensity	Viewing Angle	Forward Voltage	Unit Price
KP1608CGCK	Kingbright	Green	0603	30 mcd	120°	2.1 V	$0.09
KP1608SYCK	Kingbright	Yellow	0603	20 mcd	120°	2.0 V	$0.09
KP1608SURC	Kingbright	Red	0603	40 mcd	120°	2.0 V	$0.10
LTST C191KGKT	Lite-On	Green	0603	25 mcd	120°	2.1 V	$0.07
LTST-C191KSKT	Lite-On	Yellow	0603	15 mcd	120°	2.0 V	$0.07
LTST-C191KRKT	Lite-On	Red	0603	30 mcd	120°	2.0 V	$0.08
LS R976	OSRAM	Red	0805	80 mcd	120°	2.0 V	$0.12
LY R976	OSRAM	Green	0805	60 mcd	120°	2.0 V	$0.12
LG R976	OSRAM	Orange	0805	50 mcd	120°	2.1 V	$0.12

Table 16: LED Comparison Table

3.8.2.4. LED Selection Justification

The OSRAM LS R976 series was selected as the primary LED indicator family for this system due to its superior luminous output, mechanical resilience, and reliability in varying environments. Its 50 - 80 mcd brightness and 120° viewing angle ensure that visual signals remain clearly visible even through helmets, face shields, and smoke, addressing one of the most critical ergonomic challenges faced by first responders in the field. The larger 0805 package provides increased luminous intensity and mechanical durability compared to smaller 0603 alternatives, making it more suitable for mission critical alerts where visibility cannot be compromised.

When compared to the Kingbright KP-1608CGCK or Lite-On LTST-C191KSKT options, which offer lower brightness overall and smaller package sizes optimized for compact electronics, the OSRAM LEDs provide significantly better visibility in low visibility or high glare conditions and superior resistance to

vibration and shock. Their viewing angle also reduces the need for precise device alignment, which is particularly valuable in dynamic search and rescue scenarios where the operator's attention is focused elsewhere.

In addition, the LS R976 series is part of a broader OSRAM LED family available in multiple colors, enabling consistent mechanical, optical, and electrical performance across power, scanning, and detection indicators. This consistency simplifies design and manufacturing while maintaining uniform brightness and visibility across all system states. Together, these characteristics make the OSRAM LS R976 LEDs the most reliable and field-appropriate choice for this prototype.

3.8.3. Buzzer

3.8.3.1. CUI CMT-1203-SMT-TR

This passive piezo buzzer operates at 3 V and delivers an output of approximately 80 dB SPL, which is generally sufficient for handheld devices in noisy environments. Because it is passive, it requires a PWM signal from the MCU to generate sound, allowing full control over tone frequency and pattern. This flexibility is a major advantage in field devices that must communicate multiple system states. Its small surface mount footprint and low power draw make it suitable for portable, battery powered applications. Compared to active buzzers, the CMT-1203-SMT-TR demands more from the control firmware but rewards that complexity with significantly greater alert versatility.

3.8.3.2. Mallory Sonalert PS-1206P

This active piezo buzzer outputs around 85 dB SPL with a fixed 2.9 kHz tone. It is simple to implement, requiring only a DC voltage input and no PWM control, which reduces firmware complexity. However, its lack of tone modulation limits its usefulness in conveying multiple types of alerts. In an application where first responders must differentiate between power, detection, and fault states without looking at the device, this limitation is significant. Although its slightly higher sound pressure level improves audibility, its reduced signaling flexibility makes it less suitable for our multi-state system.

3.8.3.3. Soberton WT-1205

This magnetic buzzer achieves the highest output level at roughly 90 dB SPL, providing excellent audibility even in loud environments. However, this comes at the cost of increased current draw (about 60 mA) and larger physical size, which pose challenges for a compact, battery powered device. It is best suited for stationary systems or those where power availability is not a constraint. While its loudness is beneficial, the tradeoff in size and power consumption outweighs the advantages for a handheld search and rescue tool.

3.8.3.4. SameSky CMI-1295-03TH

The CMI-1295-03TH is a 12 mm diameter, through-hole magnetic audio indicator. It operates at a tone frequency of approximately 2.4 Hz and delivers a

sound pressure level of approximately 85 dB at 10 cm when rated at 3 V DC. This device has a compact footprint, standard PC-pin termination, and a 3 V drive voltage, making it an ideal choice for embedded systems that require breadboard prototyping and later PCB integration. It features an internally driven oscillator, which means the firmware needs to actively provide power to generate an audible alert, eliminating the need for PWM. This characteristic makes it a strong candidate for our UI design, where reliable loud audible feedback is essential.

Part Number	Type	Voltage	Sound Level (dB)	Control	Current Draw	Unit Price
CUI CMT-1203-SMT-TR	Passive Piezo	3 V	80	PWM (variable tones)	20 mA	1.42
Mallory PS-1206P	Active Piezo	3 V	85	Fixed	25 mA	2.75
Soberton WT-1205	Magnetic	3–5 V	90	Fixed	60 mA	1.15
CMI-1295-03TH	Magnetic	3 V	85 dB	Fixed	30 mA	1.14

Table 17: Buzzer Comparison Table

3.8.3.5. Buzzer Selection Justification

The **CUI CMT-1203-SMT-TR**, **Mallory PS-1206P**, and **Soberton WT-1205** buzzers were initially evaluated as potential audio alert components due to their compact size, low power consumption, and ability to generate clear audible tones for system notifications. Passive buzzers such as the CUI CMT-1203-SMT-TR offer the additional advantage of allowing the microcontroller to generate different tones and patterns to communicate system states without relying on visual feedback. However, during the design process it was determined that incorporating a buzzer alongside the system's audio speaker would introduce unnecessary redundancy.

Because the device already includes a speaker capable of producing alert tones and other audio signals, a dedicated buzzer would not provide additional functionality. Instead, the speaker can perform both notification and communication roles while simplifying the hardware design and reducing component count. For this reason, a standalone buzzer was ultimately not selected for the final design, and the system relies on the integrated speaker to provide all audible alerts and feedback.

3.8.4. Speaker

3.8.4.1. MAX98357A

The **MAX98357A** was selected as the primary audio amplifier for the system due to its efficient Class-D architecture, compact footprint, and seamless integration with the ESP32 through the I²S digital audio interface. By accepting digital audio

directly from the microcontroller, the MAX98357A eliminates the need for an external digital-to-analog converter, simplifying the signal chain and reducing potential sources of noise. The amplifier is capable of delivering up to approximately 3 W of output power into a 4 Ω speaker when powered at 5 V, providing sufficient volume for audible alerts in field environments. Its high efficiency and minimal external component requirements also make it well suited for portable, battery-powered systems where power consumption and board space must be carefully managed.

3.8.4.2. PAM8302

The **PAM8302** was considered as an alternative amplifier due to its compact size and ability to deliver up to 2.5 W of output power with relatively high efficiency. As a Class-D amplifier, it offers low heat generation and good power performance in small embedded systems. However, the PAM8302 requires an analog audio input, meaning a separate digital-to-analog conversion stage would be necessary when interfacing with the ESP32. This additional hardware would increase circuit complexity, board space requirements, and potential susceptibility to electrical noise. While the PAM8302 remains a reliable option for analog audio systems, its lack of native digital audio support makes it less convenient for integration with the system's digital audio architecture.

3.8.4.3. LM386

The **LM386** was also evaluated due to its widespread use in small audio amplifier applications and its simple implementation. This low-voltage Class-AB amplifier requires minimal external components and is commonly used for basic audio amplification tasks. However, compared with modern Class-D amplifiers, the LM386 is significantly less power efficient and can generate more heat during operation. It also provides lower output power and relies on an analog input signal, which again would require an external digital-to-analog conversion stage when used with the ESP32. These limitations make it less suitable for a compact, battery-powered system that requires efficient amplification of digitally generated audio signals.

Part Number	Input type	Power	Logic	Amplifier class	Efficiency	Unit Price
MAX98357A	I2S	~ 3 W	3.3 v	D	85-90%	Low
PAM8302	Analog	~ 2.5 W	5 v	D	85%	Low
LM386	Analog	~ 0.7 W	5 v	AB	50-60%	Moderate

Table 18: Speaker Comparison Table

3.8.4.4. Speaker Selection Justification

The MAX98357A was selected as the primary audio amplifier for the system due to its efficient Class-D architecture, compact footprint, and ability to directly

interface with the ESP32 using the I²S digital audio protocol. By accepting digital audio input, the MAX98357A eliminates the need for a separate digital-to-analog converter, simplifying the overall signal chain and reducing potential sources of electrical noise. Compared with alternatives such as the PAM8302, which requires an analog input stage, and the LM386, which offers lower efficiency and reduced output power, the MAX98357A provides a more streamlined and power-efficient solution for a portable embedded system. Its high efficiency also minimizes heat generation and helps extend battery life, which is important for a handheld device intended for field deployment.

To fully utilize the amplifier's output capability, the system was paired with a 3 W, 4 Ω speaker, which allows the amplifier to deliver sufficient acoustic output for alert tones. This combination provides adequate loudness for outdoor or high-noise environments while maintaining a compact size that can be easily integrated into the prototype enclosure.

3.8.5. Display Screens

3.8.5.1. ILI9341 TFT LCD

A compact and well supported display option is the 3.2″ TFT LCD using the ILI9341 controller. It offers a 240×320 resolution and supports up to 262,000 colors which is sufficient for showing multiple sensor readings, status icons, and simple graphics. The ILI9341 includes built in frame memory for smooth refreshing, reducing the workload on the ESP32-WROOM. It supports both 8 bit parallel and SPI communication, though SPI is preferred for this design because it uses fewer GPIO pins and still provides adequate update speed. The module operates at 3.3 V logic, making it directly compatible with the ESP32-WROOM without level shifting. Common libraries such as TFT_eSPI and Adafruit GFX offer full driver support. The LED backlight typically draws 80-120 mA and can be dimmed through PWM control. While this 3.2″ model is cost-effective ($20–$30), its small size limits visibility when displaying multiple data windows, and the standard version is not optimized for outdoor readability. Overall, it's a proven and straightforward option for compact GUIs.

3.8.5.2. 4.3" 800x490 RA8875 TFT LCD Module

A great option for a larger and higher quality display is the EastRising 4.3" TFT LCD module that uses the RA8875 controller. It has an 800×480 resolution, which gives a lot more screen space compared to smaller displays like the ILI9341. The built in RA8875 chip takes care of most of the display work, such as drawing shapes and handling frame refresh, which helps reduce processing load on the ESP32 WROOM. The RA8875 supports both SPI and 8/16-bit parallel communication modes. In this design, SPI is used to minimize wiring and conserve GPIO pins while maintaining sufficient update speed. The module runs at 3.3 V logic, which matches with the ESP32, and the LED backlight typically draws around 150-250 mA. The resolution and color depth make it a good choice for showing multiple readings, plots, or menu interfaces. However, since it's not

sunlight readable by default, it may require a shade or antiglare layer for outdoor use. The cost is around $40, which is reasonable for its size and capability.

3.8.5.3. 4.3” RA8875 Capacitive Touch Module

This version of the EastRising display adds a capacitive touchscreen, communicating via I2C for touch input while retaining SPI for graphics. Capacitive touch provides smoother, more responsive control similar to a tablet, making it ideal for menus or system adjustments on the screen. It operates at 3.3 V logic and draws roughly the same current as the non capacitive touch model (≈200 mA). Both SPI and I2C are natively supported by the ESP32-WROOM, so this module is fully compatible without additional hardware. The cost is around $50, representing a worthwhile upgrade if interactive touch control aligns with project goals.

3.8.5.4. 4.3” 800x490 IPS TFT LCD

The 4.3" 800×480 IPS TFT LCD module (ER-TFTMC043-3) includes a built-in RA8875 graphics controller, which makes it directly compatible with the ESP32-WROOM through the SPI interface. The RA8875 handles low level display tasks such as drawing, refreshing, and buffering image data, allowing the ESP32 to focus on other system functions like sensor processing and communication. The IPS panel provides excellent color accuracy and wide viewing angles, making it suitable for viewing from multiple positions or in handheld configurations. It operates at 3.3 V logic, so no level shifting is required, and typical current draw from the LED backlight is around 150–200 mA, which can be adjusted using PWM dimming. The 800×480 resolution offers plenty of space for UI elements such as radar readings, sensor data, and system alerts. While it is not fully sunlight readable, visibility can be improved through anti-reflective coatings or a shaded enclosure. This display is priced around $13–$20 and offers a high-quality image, built in controller convenience, and straightforward integration with the existing SPI based design.

3.8.5.5. 4.3” 740-nit Sunlight Readable TFT

Since this project requires outdoor operations and high ambient light environments, the Crystalfontz 4.3” 800x480 TFT LCD is a standout option. This module is designed for readability with a backlight rated at approximately 740 nits (a unit of luminance), significantly brighter than the standard TFT modules. The display integrates a resistive touch panel and communicates via an SPI interface when paired with a compatible driver board or adapter. The high brightness output does increase power consumption a bit, roughly 250–350 mA, but allows excellent visibility in daylight, making it ideal for field deployments or emergency response applications. While the resistive touchscreen is less advanced than capacitive options, it performs reliably in conditions where gloves or moisture are present. The module’s physical construction is robust, with a thick glass cover and extended temperature tolerance. It costs around $30 to be competitive for a display of this brightness, making it a great candidate for outdoor or portable instrumentation interfaces.

3.8.5.6. 4.3” Capacitive Touch LCD

The 4.3” IPS TFT Capacitive Touchscreen with an 800×480 resolution and SSD1963 display controller provides a bright, wide angle panel that is well suited for embedded UIs. The SSD1963 handles all timing, frame buffering, and RGB to parallel conversion, allowing the display to interface with microcontrollers like the ESP32-WROOM through an 8-bit or 16-bit parallel data bus without requiring a high-speed native RGB interface. The controller also integrates programmable backlight control, flexible display timings, and command-based configuration over a simple control bus. Because the SSD1963 offloads nearly all video signaling, this module is significantly easier to integrate into low power microcontroller systems compared to raw RGB panels. However, the parallel interface requires a substantial number of GPIO pins and careful routing on the PCB. With low power consumption and a reliable capacitive touch panel, this display is a strong choice for projects needing a responsive 4.3” UI without additional graphics hardware.

Models	ILI9341 TFT LCD	RA8875 Module	RA8875 Capacitive	Sunlight TFT	IPS with RA8875	Capacitive LCD
Display Size	3.2”	4.3”	4.3”	4.3”	4.3”	4.3”
Driver IC / Controller	ILI9341 (onboard)	RA8875 (onboard)	RA8875 (onboard + I2C touch)	SSD1963 (on board)	RA8875 (onboard)	SSD1963 (on board)
Resolution	240x320	800x480	800x480	800x480	800x480	800x480
Interface	SPI	SPI	SPI +I2C	SPI	SPI	Parallel bus + i2C
Voltage / Logic Compatibility	3.3 V (ESP32 direct)	3.3 V (ESP32 direct)	3.3 V (ESP32 direct)	3.3 V	3.3 V	3.3 V (ESP32 direct)
Power Consumption	80-120 mA (LED backlight)	150 – 250 mA typ.	160-250 mA typ.	250-350 mA (high brightness)	150 – 200 mA typ.	~200 mA typ.
Outdoor Readability	Limited	Moderate	Moderate	High	Moderate	Moderate
Ruggedization Options	Antiglare film / shade	Antireflective cover glass (optional)	Antiglare film / shade + touch overlay protection	Toughened glass / extended temp range	Matte overlay	Touch overlay
Use Case Fit	Compact, budget friendly UI prototype	Larger display for sensor dashboard and maps	Interactive UI with touch control menus	Outdoor or field use	Color rich display for data visualization	Modern touch GUI for portable devices

Table 19: LCD Part Comparison

3.8.5.7. LCD Selection Justification

The ESP32-WROOM module supports SPI and I2C communication natively, which makes displays using those interfaces the most suitable for this design. While RGB and MIPI-DSI displays such as the Waveshare 4.3″ capacitive LCD offer excellent visual quality, they require additional controller hardware or higher end microcontrollers like the ESP32-S3 to operate efficiently. Therefore, this project prioritizes SPI based modules such as the 3.2″ IPS ILI9341 Touch Display, which provides a high resolution 240x320 display, low latency SPI integration, and optional touchscreen functionality through I2C. These features make it directly compatible with the ESP32-WROOM, minimizing complexity while maintaining strong visual performance and moderate power consumption.

3.9. Processing and Control Modules

The processing and control subsystem oversees the operation, coordination, and data management of all sensing, communication, and user-interface components within the life-detection system. This module guarantees accurate capture, processing, and interpretation of real-time signals from radar, UWB, IR, and optional seismic sensors. It also manages system feedback, power control, and external communication with other rescue or monitoring platforms. To identify the most suitable controller for this application, we meticulously examined and compared various architectures, including MCUs, Microprocessors (MPUs), Systems-on-Chip (SoCs), co-processor hybrid designs, Field-Programmable Gate Arrays (FPGAs), and Single-Board Computers (SBCs). Each offers distinct advantages in terms of computational power, energy efficiency, and design intricacy.

3.9.1. Microcontroller Unit

A MCU acts as the central processing unit in the proposed life-detection system, handling all timing, sensing, and communication tasks in real-time. Microcontrollers (MCUs) combine the processor, memory, and peripherals into a compact device, making them highly efficient for embedded control applications. In this project, the MCU orchestrates data collection from various sensors, including radar, UWB, IR, and optional seismic sensors. It performs lightweight signal processing and fusion, and interfaces with the LCD and user controls. Its low power consumption, compact size, and real-time responsiveness make it an ideal choice for a portable, battery-operated search and rescue device. Among the various options considered, including the ESP32-S3, ESP32-WROOM-32, and Raspberry Pi Pico W, the ESP32 WROOM 32 stood out as the best choice. It has proven reliability, dual-core performance, integrated Wi-Fi/Bluetooth connectivity, and versatile peripheral support. It offers the most balanced combination of processing capability, communication range, and integration flexibility, aligning well with the system’s performance and power objectives.

3.9.2. Microprocessor

A microprocessor (MPU) differs from an MCU in that it relies on external memory and typically runs a full operating system, such as Linux. MPUs like the

Raspberry Pi 4, BeagleBone Black, or NXP i.MX series offer significantly higher computational throughput and memory resources, which enable advanced processing tasks such as machine-learning inference, radar image reconstruction, and real-time mapping. However, MPUs demand more power, longer boot times, and larger board space, making them impractical for compact, field-deployable devices. MPUs are an excellent platform for research purposes, such as post-processing sensor data or simulating AI-driven detection algorithms. However, they are less suitable as the primary embedded controller in a portable rescue tool due to power and ruggedness constraints.

3.9.3. System-on-Chip

A System-on-Chip (SoC) seamlessly integrates the flexibility of a MCU with the connectivity and compute power of a Multi-Processing Unit (MPU). This integration allows the SoC to combine multiple subsystems, such as wireless radios, memory, and AI acceleration, onto a single silicon die. Devices like the ESP32-S3, Nordic nRF54 series, or Qualcomm QCA4020 fall under this category. System-on-chips (SoCs) are ideal for edge computing applications as they enable onboard AI classification and sensor fusion without the need for external processors. Their high integration level reduces the overall number of components and simplifies PCB design, while also providing enhanced communication options like Wi-Fi, BLE, and Thread. SoCs offer a well-rounded solution for life detection, striking a balance between performance, power efficiency, and wireless versatility. This makes them an attractive research direction for future system iterations that prioritize real-time AI decision-making at the edge.

3.9.4. MCU and Co-Processor Architecture

In a hybrid architecture, a MCU collaborates with a co-processor or secondary core to delegate specific tasks, such as buffering sensor data, encrypting information, or managing power consumption. Examples include microcontrollers with ultra-low-power (ULP) cores, such as the ESP32’s internal ULP or the STM32U5 paired with a sensor hub. This distributed workload design enables the main microcontroller to remain in low-power mode while the co-processor manages background monitoring. This is particularly advantageous in long-duration rescue operations where battery life is of utmost importance. Co-processor setups enhance real-time performance by parallelizing tasks, such as managing continuous radar sampling while ensuring responsive user interaction. Although more intricate to program, this architecture improves system efficiency and fault tolerance, making it an advanced yet practical upgrade path for future prototypes.

3.9.5. Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) offers unparalleled flexibility through hardware-level reconfiguration, enabling the implementation of custom signal-processing pipelines for radar or seismic data. FPGAs are highly adept at

executing multiple tasks simultaneously, such as real-time Doppler filtering, Fast Fourier Transform computation, or multi-sensor synchronization. Devices like the Xilinx Artix-7 or Intel Cyclone 10 can significantly enhance latency and processing throughput compared to software-based MCUs. However, they necessitate specialized design tools and hardware description languages, which come with increased power consumption and higher costs. FPGAs are best suited for research and algorithm validation, especially in testing advanced signal-processing or AI fusion models, rather than the initial portable prototype.

3.9.6. Single-Board Computer

A Single-Board Computer (SBC), like the Raspberry Pi 4/5, NVIDIA Jetson Nano, or BeagleBone AI, serves as a comprehensive computing platform capable of executing intricate operating systems and sophisticated software environments. Supercomputers (SBCs) are valuable in research for data visualization, model training, and post-processing of field data. They provide powerful CPUs and GPUs that can support graphical UIs or cloud communication. While they provide unparalleled flexibility for development, they are larger, power-hungry, and less robust than embedded MCUs, which restricts their suitability for handheld or battery-limited devices. In this project, an SBC can function as a ground station or analysis companion, handling AI processing and remote monitoring tasks while the MCU performs low-level sensing and control.

3.10. Processing and Control Module Part Selection

A MCU was selected as the primary processing component for this system because it offers the ideal combination of performance, efficiency, and integration for an embedded life-detection device. MCUs, unlike microprocessors or single-board computers, which necessitate external memory and higher power consumption, integrate the processor, memory, and essential peripherals into a single, compact chip. This design makes them ideal for portable, battery-powered systems. Its real-time processing capability allows it to manage multiple sensor inputs simultaneously, perform lightweight signal processing, and control UI elements without latency. MCUs provide exceptional reliability, rapid startup times, and predictable performance, all of which are crucial for a search and rescue tool operating in unpredictable environments. This architecture guarantees efficient resource utilization, simplifies PCB design, and ensures long-term stability. Consequently, it makes an MCU the most practical and mission-ready choice for the core control system.

3.10.1. ESP 32

The ESP32 platform, specifically the ESP32-WROOM-32 variant, is a dual-core 32-bit Xtensa LX6 processor running up to 240 MHz with integrated Wi-Fi (802.11 b/g/n) and Bluetooth v4.2 connectivity. It provides up to 30 multiplexed GPIO pins and supports UART, SPI, I²C, PWM, ADC, and DAC interfaces, which makes it ideal for managing multiple sensors simultaneously. Its onboard wireless capability eliminates the need for external transceivers, allowing the

MCU to handle both data acquisition and wireless communication within a single compact module.

For this project, the ESP32 offers sufficient processing bandwidth to manage radar data streaming from the TI IWR6843AOP module over UART while concurrently reading temperature and CO_2 data via I²C and coordinating inertial measurements from the M5Stack M135 INS. Under the FreeRTOS environment, one core can be dedicated to radar data reception and pre-processing, while the other manages wireless transmission, display updates, and control logic. This dual-threaded structure ensures deterministic timing with minimal latency, which is critical for continuous vital-sign detection and user feedback. With a nominal 3.3 V supply and typical active current of 150–250 mA, it comfortably fits within the handheld system's 10 W power budget and supports battery operation for extended missions. The extensive Espressif SDK ecosystem (ESP-IDF and Arduino frameworks) also minimizes firmware-development time and provides stable Wi-Fi and BLE libraries suitable for field deployment.

3.10.2. STM32

The STM32 architecture, produced by STMicroelectronics, encompasses a broad family of ARM Cortex-M-based microcontrollers ranging from low-power M0 devices to high-performance M7 variants operating at up to 400 MHz. These MCUs are renowned for precise timing, extensive analog capabilities, and industrial reliability. They are widely used in control systems and high-determinism applications due to their predictable interrupt handling and hardware-level timers.

In the context of this life-detection system, the STM32 could provide accurate timing for radar triggering or synchronized sensor sampling. However, STM32 devices lack built-in wireless connectivity; Wi-Fi or Bluetooth modules must be added externally, increasing board complexity, wiring density, and power overhead. Integrating an external wireless chipset would also require additional firmware development for network communication. While the STM32's analog precision and hardware timers outperform the ESP32 for deterministic control, these advantages are less critical in this design because the radar module already incorporates its own signal-processing DSP and timing control. Additionally, STM32-based development typically demands more extensive firmware infrastructure, requiring embedded C and hardware abstraction layers, which slows prototyping relative to the streamlined ESP-IDF toolchain.

3.10.3. RasberryPi

The Raspberry Pi platform, such as the Pi Zero 2 W or Pi 4 Model B, offers general-purpose processing capability, featuring multi-core ARM Cortex-A53/A72 CPUs running between 1.2 and 1.8 GHz, gigabytes of RAM, and a full Linux operating system. These single-board computers integrate Wi-Fi and Bluetooth connectivity by default and support high level programming environments. While the processing power would easily accommodate radar FFT computation, data

fusion, and even visualization, the trade-off is significant power consumption (typically 500 mA – 2 A) and a larger physical footprint.

For a handheld, battery-powered device designed for field search and rescue operations, the Raspberry Pi's size, boot time, and power draw make it impractical. Ruggedized embedded systems require regulated 5 V input, active cooling in certain cases, and a secure file system shutdown to prevent data corruption. These features are not compatible with the instantaneous and reliable operation of ruggedized embedded systems under field stress. While the Pi could serve as a development testbed for advanced radar-signal visualization or machine-learning experiments, it exceeds the computational needs of this prototype and compromises energy efficiency and portability.

Platform	Processor	Wireless	Power (Active)	Key Advantages	Limitations
ESP32	Dual-core Xtensa LX6 @ 240 MHz	Wi-Fi b/g/n + BT 4.2	150–250 mA @ 3.3V	Integrated wireless, low cost, compact, proven SDK support	Moderate DSP power, limited ADC precision
STM32	ARM Cortex-M4/M7 up to 400 MHz	None (external module required)	80–150 mA @ 3.3V	Precise timing, strong analog control, industrial reliability	No built-in wireless, more complex development
Raspberry Pi	Quad-core ARM Cortex-A53/A72 @ 1.2–1.8 GHz	Wi-Fi + BT onboard	500 mA–2 A @ 5V	High performance, full OS, easy ML/visualization	High power draw, large size, slow boot

Table 20: MCU comparison

3.10.4. MCU Selection Justification

After a comparative evaluation, the ESP32 was selected as the primary microcontroller architecture for the handheld system. It delivers the most balanced combination of performance, wireless integration, cost, and power efficiency. The module's dual core structure, native Wi-Fi/Bluetooth support, and extensive peripheral compatibility make it uniquely suited to act as the bridge between the radar sensing front end and the UI or network layer. While the STM32 provides excellent precision and reliability, it introduces unnecessary complexity and lacks built-in communication capability. The Raspberry Pi, though far more powerful, would drastically reduce battery life and increase hardware overhead beyond the system's design envelope. The ESP32 therefore stands as the most practical, energy efficient, and integration-ready option for this stage of development, enabling the project to meet real-time performance goals without exceeding cost or power constraints.

3.11. MCU Part Comparison

3.11.1. ESP32-WROOM-32

The ESP32-WROOM-32 is a compact, dual-core wireless microcontroller module developed by Espressif Systems and widely used for IoT and embedded sensing applications. It integrates a 32-bit Xtensa LX6 CPU running up to 240 MHz, a ULP RISC co-processor for low-power background operation, 520 KB SRAM, and 4 MB SPI flash. The module supports 2.4 GHz Wi-Fi (802.11 b/g/n) and Bluetooth v4.2 (Classic + BLE), enabling short- and medium-range communication for field devices. It operates within a nominal voltage range of 3.0 V to 3.6 V, with a typical load current of 150–250 mA. In deep-sleep mode, it drops to microamp levels, making it suitable for portable, battery-powered systems.

In this project, the ESP32-WROOM-32 serves as the central processing and communication hub, connecting the radar, IR, and inertial sensors. Its abundant peripheral set, up to 30 GPIO pins multiplexed for UART, SPI, I2C, PWM, ADC, and DAC, provides sufficient flexibility to connect all sensing modules simultaneously without the need for additional expanders. The 240 MHz dual-core processor provides enough computational headroom for moderate signal-processing tasks such as filtering radar outputs, decoding sensor data, or running control logic under FreeRTOS. Wi-Fi and BLE support allow wireless data transfer between rescuers' devices or to a base station, aligning with the project's goal of producing a networked, field-deployable life-detection tool.

The ESP32-WROOM-32 combines high processing power, built-in wireless communication, and low cost in a single, well-supported package. Its dual-core architecture allows one core to manage sensor acquisition and the other to handle communications or display updates, improving real-time responsiveness. The integrated security features, such as, AES, RSA, ECC encryption, and secure boot, make it suitable for scenarios requiring data integrity and secure firmware updates. The development ecosystem is mature, with support from Espressif IDF, Arduino IDE, and PlatformIO, which reduces firmware-development time. Its power efficiency and small footprint also support handheld or wearable implementation without external computer hardware.

Despite its versatility, the ESP32-WROOM-32 is not optimized for heavy digital signal processing. Operations such as FFT-based range-Doppler analysis or high-rate radar filtering can saturate its cores if implemented entirely in software, potentially requiring off-loading to the radar's onboard DSP or external processors. The ADC performance is limited to 12 bits and can suffer from noise, restricting precision analog measurements. Additionally, the module supports only 2.4 GHz Wi-Fi, which can be congested in some environments, and its wireless range may be limited in debris-dense or metallic areas typical of search-and-rescue zones. Careful power-management design is also necessary to ensure long runtime from portable batteries, as active-mode currents near 250 mA can drain smaller cells quickly.

Overall, the ESP32-WROOM-32 offers an excellent balance of processing capability, integration, and connectivity for the prototype stage of this search-and-rescue system. It is powerful enough to manage sensor coordination and wireless communication while remaining compact, inexpensive, and easy to program. However, future iterations aiming for advanced radar processing or extended field endurance may require supplementing it with a dedicated signal-processing coprocessor or higher efficiency power system.

3.11.2. ESP32-S3-WROOM-1

The ESP32-S3-WROOM-1 is an upgraded wireless MCU module from Espressif designed for applications requiring enhanced AI acceleration, higher memory capacity, and improved peripheral interfaces. It features a dual-core Xtensa LX7 processor that operates at up to 240 MHz, providing higher efficiency compared to the LX6 cores in the original ESP32-WROOM-32. Additionally, it incorporates vector (SIMD) instructions that enhance the processing of signals and neural networks. The module typically integrates 8 MB PSRAM and 4–16 MB flash, providing ample buffer space for radar and sensor data. Wireless support includes 2.4 GHz Wi-Fi (802.11 b/g/n) and Bluetooth 5 (BLE + Bluetooth Mesh), with longer range and higher throughput than previous generations. Operating voltage remains 3.0 V to 3.6 V, and active-mode power consumption is about 200–240 mA, like the WROOM-32, while sleep currents remain in the microamp range.

For this project, the ESP32-S3-WROOM-1 would serve as a strong alternative to the WROOM-32, especially if on-device radar processing or machine-learning-based signal classification (such as identifying respiration patterns) becomes a design priority. Its vector instructions and expanded PSRAM make it capable of handling heavier FFT computations, filtering, or data-fusion tasks locally rather than relying entirely on the radar's internal DSP. The larger flash and RAM are particularly useful for implementing image- or heatmap-based output from the IR sensor or logging multi-sensor data for post-event analysis.

Compared with the ESP32-WROOM-32, the S3 offers noticeably better computational throughput for signal-processing tasks and more memory for buffering or ML inference. Its Bluetooth 5 and BLE Mesh support extend wireless range and improve device-to-device connectivity, valuable for multi-rescuer coordination. The S3 also introduces USB OTG, eight capacitive touch inputs, and an improved ADC (13-bit effective), expanding the device's interface options for custom controls or diagnostics. Additionally, its enhanced security features, including a digital signature, HMAC, and secure boot v2, further fortify the device against firmware tampering.

The S3 is slightly more expensive and less power-efficient than simpler ESP32 modules when running AI or high-throughput tasks continuously. Its 2.4 GHz-only radio still limits range and penetration through rubble compared with sub-GHz systems, and the expanded memory footprint increases wake-up current slightly in low-power modes. While Espressif's SDK fully supports the S3, libraries for some peripherals (e.g., USB or camera interfaces) are larger and can increase

firmware complexity. Overall, the ESP32-S3-WROOM-1 represents a balanced upgrade for this system, offering superior processing headroom for future radar-signal enhancement or on-board ML inference while maintaining pin-level compatibility and similar power requirements to the original ESP32. For a prototype that may evolve into a more autonomous, AI-aided platform, the S3 is the more forward-looking choice.

3.11.3. ESP32-C6-WROOM-1

The ESP32-C6-WROOM-1 is Espressif's latest generation single-core RISC-V wireless MCU module, emphasizing efficiency, advanced connectivity, and extended-range wireless features rather than raw processing power. It runs a 160 MHz RISC-V core and includes a ULP coprocessor for sensor polling during deep sleep. Unlike its predecessors, this model supports dual-band connectivity, including 2.4 GHz Wi-Fi 6 (802.11 ax) and Bluetooth 5 LE (with 802.15.4 for Thread and Zigbee networks). This makes it one of the most versatile MCUs currently available. The module combines 4 MB of flash memory and 400 KB of SRAM, operates at a voltage range of 3.0 to 3.6 V, and achieves significantly reduced active power consumption (typically between 80 and 120 mA) with microamp-level deep-sleep current. This makes it an ideal choice for systems that require extended battery life.

In the context of this project, the ESP32-C6-WROOM-1 offers strong advantages where low power and robust network communication take priority over heavy local computation. Its Wi-Fi 6 radio supports Target Wake Time (TWT) and improved spectral efficiency, extending battery life and maintaining stable connectivity in noisy, crowded environments such as urban disaster zones. The inclusion of Thread and Zigbee technology allows for mesh networking between multiple handheld devices, which is an attractive feature for coordinating search teams without the need for external access points.

The C6 delivers next-generation wireless performance and exceptional power efficiency, potentially doubling battery runtime compared to the WROOM-32. Its Wi-Fi 6 and Thread/Zigbee support introduce robust mesh networking options for device-to-device communication, and the RISC-V architecture benefits from modern toolchain support and smaller firmware size. The reduced current draw and advanced sleep scheduling align well with long-duration field missions where recharging is limited.

The C6 sacrifices raw computational capability, its single 160 MHz core is considerably slower for heavy DSP or FFT workloads than the dual-core LX6/LX7 variants. It also lacks the large PSRAM expansion found on the S3, limiting its ability to buffer high-rate radar data or run complex fusion algorithms. As a newer platform, some peripheral drivers (especially for mixed SPI/I²C high-speed sensors) are less mature than the S3's ecosystem.

Overall, the ESP32-C6-WROOM-1 is an excellent candidate for future revisions of the project where extended battery life, mesh networking, and secure wireless operation are prioritized over intensive on-board processing. For the current

prototype—where real-time radar data handling and local computation remain key, the C6 would likely serve better as a low-power communication node rather than the primary processing MCU.

Module	CPU	Wireless	Power (Active)	Key Advantages	Limitations
ESP32-WROOM-32	Dual-core LX6 @ 240 MHz	Wi-Fi b/g/n + BT 4.2	150–250 mA	Proven, low-cost, strong I/O, easy integration	Older wireless standard, limited DSP
ESP32-S3-WROOM-1	Dual-core LX7 @ 240 MHz	Wi-Fi b/g/n + BT 5	200–240 mA	Higher performance, extra RAM, AI/DSP support	Slightly higher cost and power
ESP32-C6-WROOM-1	Single-core RISC-V @ 160 MHz	Wi-Fi 6 + BT 5 + Zigbee	80–120 mA	Very low power, mesh networking	Lower processing power, limited memory

Table 21: MCU model comparison table

3.11.4. MCU Final Selection Justification

After evaluating the ESP32-WROOM-32, ESP32-S3-WROOM-1, and ESP32-C6-WROOM-1, the team chose the ESP32-WROOM-32 as the primary microcontroller for this project. This decision was based on its optimal combination of processing power, power efficiency, hardware maturity, and ease of integration. While newer variants like the S3 and C6 offer incremental improvements in AI acceleration and wireless range, the WROOM-32 provides all the essential functionality required for the current system at a fraction of the complexity and cost. Its dual-core Xtensa LX6 processor running at 240 MHz delivers sufficient computational power to handle real-time data acquisition and preprocessing from the radar, IR, and inertial sensors without the need for external coprocessors. The module's extensive peripheral set, comprising three UARTs, multiple SPI and I2C buses, and up to 30 GPIO pins, facilitates the simultaneous operation of all sensing subsystems. This enables seamless communication with the radar module over UART while maintaining stable I2C and SPI channels for the other peripherals.

In the context of this life-detection and localization system, the ESP32-WROOM-32 offers proven reliability and an extensive development ecosystem, both critical for rapid prototyping and field deployment. Its integrated Wi-Fi and Bluetooth connectivity allow the device to transmit sensor data wirelessly to mobile devices or command centers, fulfilling the project's requirement for remote monitoring in search-and-rescue environments. The availability of FreeRTOS under the Espressif IDF framework ensures efficient multitasking, allowing one CPU core to manage radar data streaming while the other handles wireless communication, sensor fusion, or display updates. Power consumption remains moderate, typically between 150 mA and 250 mA during active sensing, making it feasible for battery powered operation while maintaining the responsiveness required for real-time detection.

Although the ESP32-S3 and C6 offer newer architectures, their advantages are not critical to this prototype's success. The S3's additional AI acceleration and memory are underutilized in a system where radar signal processing is offloaded to the IWR6843AOP's onboard DSP, and the C6's single core design sacrifices too much processing capability for marginal gains in network range and efficiency. The WROOM-32, by contrast, provides a stable, well-documented, and field-tested platform that integrates cleanly with the chosen radar and sensor modules. It satisfies all electrical, communication, and performance requirements for the prototype phase, ensuring dependable operation while keeping hardware costs and development time low. For these reasons, the ESP32-WROOM-32 is the most practical and technically appropriate microcontroller for the current stage of the project.

3.12. Power Supply

Proper power management is critical to ensure stable and efficient operation of the system. Each subsystem, the ESP32-WROOM microcontroller, the 4.3″ ILI9341-based TFT LCD display, and the Infineon Radar Baseboard MCU7, requires precise voltage regulation and power budgeting to maintain consistent performance without overloading the supply. This section discusses the power requirements and distribution across the system, detailing how each component draws current under normal operating conditions and how low-power peripherals are incorporated to improve efficiency.

3.12.1. MCU7 Baseboard

The Infineon MCU7 radar baseboard serves as the main interface and power distribution platform for the IWR6843AOP radar module, providing regulated 3.3 V and 5 V outputs to the radar, ESP32-WROOM microcontroller, and other peripherals. According to Infineon's documentation, the MCU7 operates from a 5 V input and distributes power through onboard low-noise regulators to support digital logic, analog front-end circuits, and radar transceiver functions. The total active power consumption of the IWR6843AOP radar mounted on the MCU7 board is approximately 2.0 – 2.5 W, with an additional 0.3 – 0.5 W allocated to the ESP32 and communication interfaces. The system therefore requires a stable 5 V, 600 mA minimum supply, with additional current margin to account for startup surges and simultaneous radar and LCD activity. The MCU7 includes onboard voltage sequencing to ensure that the 3.3 V logic rail is enabled before the radar analog front-end, preventing latch-up or transient errors during power-up. These characteristics define the primary constraints for regulator selection and overall power budgeting in the system design.

3.12.2. LCD Display

The ER-TFTM043A1-7S TFT LCD serves as the primary UI for the system, displaying real-time data, status indicators, and user prompts. Because the display is one of the more power-intensive components, it's important to evaluate its electrical and operating requirements carefully. Since the ESP32-WROOM

uses a 3.3 V logic system and an Intel 8080 style parallel bus, the display will be ordered with the 8080 interface and 3.3 V power configuration. The backlight can be controlled by the onboard SSD1963 controller (J4 short, J3 open) or through an external signal if reversed. At 3.3 V operation, the module typically draws around 450 mA, though this can vary with brightness and refresh activity. The backlight consumes the most power, so it should be budgeted for worst case brightness and scaled down proportionally if dimming with PWM. Additional peripherals such as the touch controller add roughly 3-10 mA, and logic lines stay within 3.3 V compatibility for direct use with the ESP32-WROOM. When combined with the ESP32's own current demand (about 240-300 mA peak during Wi-Fi activity), a total 3.3 V rail capacity of around 1 A is recommended for safe headroom. This ensures stable operation during current spikes and allows flexibility for added peripherals or higher brightness levels.

3.12.3. ESP32-WROOM

The ESP32-WROOM is the main microcontroller and communication module of the system, responsible for processing sensor data, managing peripheral interfaces, and handling Wi-Fi/Bluetooth communication. According to the Espressif ESP32-WROOM-32 datasheet v3.5, the module operates at 3.3 V with current draw varying Wi-Fi mode and transmission power. According to the ESP32 Series Datasheet v5.1 [1], typical transmit currents are about 180 mA for 802.11n, 190 mA for 802.11g, and up to 240 mA for 802.11b at +19.5 dBm. Receive current averages around 95-100 mA. Although only one RF mode is active at a time, the highest typical draw of 240 mA (~ 0.79 W) is used for design to ensure supply stability during transmit peaks. A 25% margin is added, giving a recommended supply capacity of roughly 300 mA. In low-power modes, current decreases to 20-25 mA in modem sleep, 0.8 mA in light sleep, and 100-150 µA in deep sleep.

3.12.4. Inertial Navigation System

The custom INS stack consists of an MPU9250 9-axis IMU (accelerometer, gyroscope, and magnetometer) and a BMP280 barometric pressure sensor, both communicating over the shared I²C bus at 3.3 V. Combined steady-state current draw is approximately 36 mA at 3.3 V (~0.12 W), with the MPU9250 accounting for roughly 3.5 mA in full 9-axis operation and the BMP280 contributing under 1 mA in normal mode; the remainder of the allocation covers I²C bus activity, Madgwick filter polling overhead on the ESP32, and periodic calibration reads. The 3.3 V power budget allocates 40 mA for this subsystem, providing margin for gyroscope and accelerometer bias calibration bursts (300-sample averages collected at startup and on START press) and barometric altitude zeroing sequences.

The MPU9250 supplies high-rate acceleration and angular velocity data for dead-reckoning and Madgwick orientation filtering, with gyro Z-axis integration maintained separately for yaw to avoid Madgwick yaw drift in magnetically disturbed indoor environments. The BMP280 provides

barometric altitude for Z-axis correction in the 8-state Extended Kalman Filter and supplies an ambient temperature reference that is fused directly into the MLX90640 thermal background model to scale adaptation rates across varying deployment environments. Both devices share the primary I²C bus (GPIO21 SDA, GPIO22 SCL) with the MLX90640 thermal camera and CAT9555 GPIO expander, and should be tied to the primary 3.3 V regulator domain with local decoupling (≥10 µF bulk + 0.1 µF ceramic close to each supply pin) and included in the always-on load group.

3.12.5. Ultra-Wide Band

The DWM3000EVB integrates Qorvo's DWM3000 UWB module (DW3000/DW3110 transceiver). While the module brief notes low power consumption, numeric current values are specified in the DW3000 IC datasheet, which provides TX/RX current profiles. Under active ranging, the DW3000 exhibits burst currents with RX peaks on the order of 70-80 mA and TX peaks ~20-35 mA, depending on supply rails and mode; for power budgeting we allocate a worst-case transient of ~90 mA at 3.3 V (~0.30 W). In sleep modes, current falls to sub-milliamp/µA levels. The EVB exposes a J1 current measurement jumper to verify module draw in system, once the component has been delivered, the measured values from this will be used for final power budgeting.

3.12.6. Infrared Sensor

The MLX90640 is a 32×24-pixel IR thermal imaging sensor manufactured by Melexis. It operates from a 3.3 V supply with a typical current consumption of 20 mA and a maximum of 25 mA, corresponding to an estimated power draw between 0.066 W and 0.082 W. Communication is handled through an I2C interface compatible with both 3.3 V and 5 V logic, making it suitable for use with common microcontrollers such as the ESP32. The sensor's low power requirement allows it to run efficiently in embedded applications while maintaining accurate non-contact temperature measurements across its 55° field of view.

3.12.7. Low Power Supporting Components

In addition to the primary subsystems, several low power supporting components are used throughout the design to provide efficient operation and system stability. These include onboard voltage regulators, indicator LEDs, logic level shifters, and sensor interface circuits that operate within the 3.3 V domain. The regulators ensure a consistent supply to all modules while maintaining low quiescent current, minimizing overall standby consumption. Indicator LEDs draw only a few milliamps each and are typically duty cycled or turned off during idle periods to reduce unnecessary load. Level shifting ICs and I2C pull ups operate in the microamp range during static conditions, contributing negligible power usage compared to major components such as the LCD and radar module. Collectively,

these low power peripherals enable reliable monitoring and control without significantly impacting the system's total power budget, supporting energy efficient embedded operation.

Component	Voltage (V)	Current (mA)	Power (W)
ESP32-WROOM	3.3	80-300	0.26-0.99
LCD Display	3.3	~ 220	0.72
Radar Baseboard MCU7	5	400 - 500	2.0 - 2.5
IR Sensor	3.3	20-25	0.066-0.082
UWB	3.3	~90	0.3
INS	3.3	36	0.12
Supporting Components	3.3	< 50	0.17

Table 22: Power Distribution Table

3.13. Power Supply Part Selection

3.13.1. Battery Types

For this project, several rechargeable battery chemistries were evaluated to determine the most effective option for powering our device. The main selection criteria included energy density, weight, cost, safety, discharge capability, and ease of integration. The following subsections summarize each candidate chemistry and its suitability for a compact, portable radar-sensor platform.

3.13.1.1. Lithium-Ion (Li-ion)

Lithium-ion batteries are widely used in modern electronics due to their high energy density, low weight, and long service life. Their ability to store substantial energy in a small form factor makes them ideal for portable embedded systems where space is limited. They also offer fast charge cycles and relatively low self-discharge rates. However, Li-ion cells require a battery-management system (BMS) for safe charging and protection against over-voltage or short circuits. If damaged or improperly handled, Li-ion batteries may experience thermal runaway or fire, which introduces additional design and safety considerations.

3.13.1.2. Lithium-Polymer (LiPo)

Lithium-polymer batteries are a subtype of Li-ion chemistry that use a gelled polymer electrolyte instead of a liquid one. This allows them to be manufactured in thin, lightweight, and flexible enclosures, which is advantageous for embedded applications that demand compactness. LiPo packs can deliver high discharge currents suitable for powering both the ESP32 and peripherals such as the LCD backlight and radar baseboard. While they share the same safety challenges as Li-ion batteries, proper balancing and protection circuitry significantly reduce these risks. LiPo batteries were therefore considered a strong candidate due to their power-to-weight ratio and ease of sourcing in custom capacities.

3.13.1.3. Nickel-Cadmium (NiCd)

NiCd batteries are durable and capable of handling moderate discharge rates, but they have a low energy-to-weight ratio and suffer from a pronounced memory effect, reducing usable capacity over time. Their high self-discharge rate makes them unsuitable for systems that operate intermittently or remain idle between uses. In addition, cadmium is a toxic heavy metal that presents environmental disposal concerns, making this chemistry less desirable for modern embedded systems.

3.13.1.4. Nickel-Metal Hydride (NiMH)

NiMH cells are safer and more environmentally friendly than NiCd, but they still have lower energy density compared with lithium-based options. They also discharge more quickly when not in use and exhibit voltage drops under heavy loads. For this project, the combination of continuous wireless communication, sensor operation, and display output would cause NiMH packs to deplete rapidly, reducing system runtime and efficiency. Although they offer stability and low risk of thermal events, their performance limitations made them impractical for this design.

3.13.1.5. Sealed Lead-Acid (SLA)

SLA batteries are valued for their reliability, affordability, and ability to supply high surge currents, but they are considerably heavier than other types. Their low energy-to-weight ratio and bulky form factor make them unsuitable for lightweight portable systems. SLA chemistry remains common in stationary or large-scale applications but would significantly increase the overall mass of this handheld device. Despite their robust safety record and simple charging requirements, SLA batteries were excluded due to weight and volume constraints.

3.13.2. Battery Selection and Justification

Among the battery types evaluated, each presented distinct advantages and disadvantages when considered for our embedded radar-based sensing system. Lithium-ion and Lithium-Polymer (LiPo) batteries provide high energy density and lightweight performance, but come with elevated safety risks, including the potential for overheating and combustion under improper handling. Due to these concerns and faculty restrictions, LiPo batteries were excluded from consideration. Nickel-Cadmium (NiCd) and Nickel-Metal Hydride (NiMH) batteries, while durable and relatively stable, suffer from high self-discharge rates, limited energy density, and memory effects, making them unsuitable for long-duration portable applications. Sealed Lead-Acid (SLA) batteries remain a cost-effective and safe option, but their large mass and low energy-to-weight ratio make them impractical for this compact, handheld system.

After comparing each battery, the Lithium-Ion was selected as the optimal choice for this project. Li-ion cells offer a strong balance of energy density, efficiency, and safety when paired with an integrated battery management system (BMS). The BMS safeguards against overcharging, short circuits, and thermal events, addressing many of the common safety issues associated with lithium-based designs. In addition, Li-ion batteries are widely available, have long cycle life, and support high discharge currents, sufficient for powering the ESP32-WROOM microcontroller, TFT LCD display, Infineon radar module, and supporting sensors simultaneously.

Although slightly more expensive than NiMH or SLA options, the superior performance-to-weight ratio, portability, and compatibility with standard 3.7 V power regulation circuits make Lithium-Ion batteries the most appropriate power source for this design. A single-cell 3.7 V Li-ion pack rated at approximately 2500–3000 mAh provides sufficient energy to sustain the system's estimated 3-4 W load for roughly 2–2.5 hours of continuous operation, depending on sensor activity and backlight brightness. This configuration ensures a reliable and efficient power solution while maintaining safety and compliance with laboratory standards.

3.13.3. Voltage Regulation

Voltage regulation is a critical aspect of the power management system, ensuring that all components, including the ESP32-WROOM microcontroller, LCD display, radar baseboard, and sensors, receive a stable and reliable supply voltage. Regulators maintain consistent voltage levels even as the battery voltage changes during discharge, preventing fluctuations that could cause instability or damage to sensitive electronics.

In this project, the system is powered by a 3s 18650 Lithium-Ion battery protected by a Battery Management System (BMS). The BMS safeguards the cell against overcharging, deep discharging, and short circuit conditions, while the voltage regulators provide precise voltage conversion for each subsystem. Two regulated outputs are required: 3.3 V for the ESP32, LCD, and low power sensors, and 5 V for the IWR6843AOPEVM and other peripherals requiring higher voltage.

The 3.3 V rail supplies logic and communication circuitry and must deliver up to approximately 1 A during peak operation. The 5 V rail supports the radar module and other peripherals, requiring around 1 A of available current capacity. Both rails will be derived from the 3.7 V battery using high-efficiency switching buck-boost converters, capable of maintaining stable output even as the battery voltage drops during operation.

To maximize efficiency and runtime, voltage regulators with efficiencies above 80% were selected. This minimizes energy that is lost to heat and extends the usable battery life for portable operation. In addition, regulators with low quiescent current were prioritized to reduce standby power draw when the

system is idle. Proper thermal management and copper pour areas on the PCB will further help dissipate any residual heat and ensure consistent long-term performance.

The system is designed to be fully rechargeable by removing the batteries and charging via a commercial li-ion battry charger. A Battery Management System (BMS) sits between the battery and load, providing continuous protection against over-charging, over-discharging, and short circuit conditions. There are also battery fuses between the battery lines and the BMS, as well as a load fuse between the BMS and the device, which protects against any unwanted high current draws. Together, this setup ensures safe, efficient charging, and reliable operation.

3.13.3.1. LM34936

The LM34936 is a synchronous four-switch buck-boost controller designed for single inductor step-up/step-down DC-DC converters. It operates from 4.2 - 36 V (42 V absolute max) and can regulate outputs either above or below the input, with a flexible 0.8 - 30 V output range. The device uses current mode control, supports programmable switching frequency from ~100 kHz to 1 MHz, and includes advanced features like cycle by cycle current limiting, soft start, under-voltage lockout and over-voltage protection.

3.13.3.2. S7V8F3

The Pololu S7V8F3 is a compact step-up/step-down (buck-boost) switching regulator that provides a fixed 3.3 V output from a 2.7 - 11.8 V input range. It can both boost and buck a single cell Li-ion battery, which is perfect when the pack moves above and below 3.3 V as it discharges. The module is rated for up to about 1 A output at 3.3 V with typical efficiencies above 90% for lithium-battery voltages and includes built-in protections such as reverse-voltage protection and short-circuit limiting.

3.13.3.3. LM33620

The LM34936 is a synchronous four-switch buck-boost controller (no integrated power FETs) designed for single-inductor step-up/step-down DC-DC converters. It operates from 4.2 - 30 V (42 V absolute max) and can regulate outputs either above or below the input, with a flexible 0.8 - 30 V output range. The device uses current-mode control, supports programmable switching frequency from ~100 kHz to 1 MHz, and includes advanced features like cycle-by-cycle current limiting, soft start, under-voltage lockout, over-voltage protection, and optional spread-spectrum dithering for EMI reduction.

3.13.3.4. LM3478

The LM3478 is a low side N-channel MOSFET controller designed for boost topologies. It operates over a 2.97 - 40 V supply range and uses current-mode

control with an adjustable switching frequency from 100 kHz to 1 MHz. Because it only provides the control and gate drive (up to ~1 A peak), the achievable output current is set by the external MOSFET, inductor, and overall design. The part includes cycle by cycle current limit, thermal shutdown, and a power-saving shutdown mode that drops supply current to about 5–10 µA.

3.13.3.5. LM2576

The LM2576 is a monolithic step-down (buck) switching regulator with an integrated power switch, capable of delivering up to 3 A of load current. It operates over a 4 – 40 V input voltage range and is available in fixed output versions (3.3 V, 5 V, 12 V) as well as an adjustable variant. The device uses a fixed switching frequency of 52 kHz, which simplifies design but requires larger external components compared to higher-frequency regulators.

The LM2576 includes internal frequency compensation, cycle-by-cycle current limiting, thermal shutdown, and safe-area protection, making it robust and easy to implement. Due to its relatively low switching frequency, it tends to exhibit higher output ripple and larger inductor/capacitor requirements, but benefits from simplicity and strong load-driving capability. Typical quiescent current is on the order of a few milliamps, with a low-power shutdown mode available in certain variants.

3.13.3.6. LM2731

The LM2731 is an integrated boost converter with an internal 22 V FET switch and fixed-frequency operation at either 600 kHz or 1.6 MHz depending on the variant. It runs from a 2.7 - 14 V input and can switch up to about 1.8 A peak current, supporting boosted outputs up to around 20 V for moderate loads. Typical quiescent current is around 2 mA, with a very low shutdown current under 1 µA when the SHDN pin is pulled low.

Parameter	LM34936	LM2576	LM33620
Type	Buck-Boost	Buck	Buck (Integrated)
Input Voltage (V)	~4.2 - 30	4 - 40	~4.2 - 30
Output Voltage (V)	Adjustable	Adjustable	Adjustable
Max Output Current (A)	~3	3	3
Efficiency	≥ 90%	~85%	≥ 90%

Parameter	LM34936	LM2576	LM33620
Quiescent Current	25 µA	~5 mA	~ 27µA
Thermal Performance	Low heat	Moderate – high heat	Low heat

Table 23: 3.3 V Regulator Comparison

Parameter	LM3478	LM2576	LM2731
Type	Boost (Controller)	Buck	Boost (Integrated)
Input Voltage (V)	2.9 - 40	4 - 40	2.7 - 14
Output Voltage (V)	Fixed (5.0)	Adjustable	Adjustable
Max Output Current (A)	1 – 2	3	< 1
Efficiency	82 - 85%	~85%	75 - 85%
Quiescent Current	100 µA	~5 mA	16 µA
Thermal Performance	Moderate heat	Moderate – high heat	Moderate – high heat

Table 24: 5 V Regulator Comparison

3.13.3.7. Regulator Selection and Justification

Among the voltage regulator options evaluated, each presented distinct tradeoffs when considered for the portable embedded sensing system. The LM34936 offers the most flexibility as a synchronous buck-boost controller capable of both step-up and step-down operation across a wide input range. However, because it requires multiple external power components (MOSFETs, compensation network, and careful layout), it introduces significant design complexity, increased development time, and higher risk of instability if not properly tuned. The LM3478, while simpler, is limited to boost-only operation and also relies on external switching components, placing the burden of current handling and efficiency on the surrounding design. The LM2731 provides a compact, integrated boost solution with high switching frequency and small passive components, but its output current capability is limited and it cannot step down voltage, making it unsuitable for systems powered by a multi-cell battery where the input voltage may exceed the desired output.

After comparing each regulator, the LM2576 was selected as the most appropriate choice for this design. As a fully integrated buck (step-down) regulator, it is well-suited for converting the system's battery voltage to a stable 5 V rail required by the ESP32, display, and supporting electronics. Its ability to deliver up to 3 A of output current provides substantial margin for peak load conditions, ensuring reliable operation even when multiple subsystems are active simultaneously. Unlike controller-based solutions, the LM2576 requires minimal

external components and does not require complex compensation design, significantly reducing implementation risk and simplifying PCB layout.

Although the LM2576 operates at a relatively low switching frequency (52 kHz), which results in larger passive components and moderate efficiency compared to modern high-frequency regulators, these drawbacks are acceptable within the scope of this project. The increased component size is manageable within the enclosure, and the efficiency remains sufficient for battery-powered operation without excessive thermal concerns when properly designed. Additionally, its robust internal protections—including current limiting and thermal shutdown—enhance system reliability and safety.

Overall, the LM2576 provides the best balance of simplicity, current capability, and reliability for this application. Its ease of implementation, strong load-driving capability, and proven performance make it an ideal regulator for establishing a stable power rail in a portable embedded system, outweighing the benefits of more complex or specialized alternatives.

3.13.3.8. Additional Required Power Hardware

Beyond the battery and regulators, the rechargeable power system we will be using in our project requires a USB-C recpetable configured as a 5V sink with pull down resistors, two 5.1kΩ CC, a single cell Li-ion charger IC and a 1S BMS protection board. To prevent backfeed, we will also need to include ESD/TVS on VBUS, an input filter and a small load switch or ideal diode device. Each DC-DC requires it is specific inductors and input/output capacitors per their respective datasheets as well as a short feedback network. Lastly, we will need a battery holder and connector. During the prototyping phase, the system can be safely powered using a variable DC power supply instead of a battery. This allows precise control of both voltage and current while testing individual subsystems such as the ESP32, radar baseboard, and LCD display. The supply can be set to 4.0–4.2 V to simulate a fully charged single-cell Li-ion battery, or 5.0 V to represent the USB-C input voltage. A current limit of 1–1.5 A is recommended to protect components during early development and troubleshooting. The power supply connects directly to the system rails through the breadboard or test points, allowing easy measurement of voltage and current draw. This setup provides a stable and adjustable source for verifying circuit performance and regulator functionality before integrating the Li-ion battery, charger, and BMS.

3.14. Communication Protocols

The 2HADRAD system utilizes a well-structured embedded firmware framework, developed in C/C++ using one of the environments. This framework enables real-time coordination of multiple sensing modules, communication protocols, and UI components operating concurrently under the FreeRTOS task scheduler. Each module, from the radar and UWB sensors to the thermal array and INS, necessitates distinct initialization, data acquisition, and filtering procedures. The firmware manages data flow across UART, SPI, and I2C buses, ensuring

deterministic timing and low power consumption. The system architecture employs a modular approach, with independent tasks for sensor handling, data fusion, and display updates, facilitating simplified debugging and future expansion.

3.14.1. Radar Module (TI IWR6843AOP)

The radar subsystem utilizes the Texas Instruments IWR6843AOP millimeter-wave radar, which outputs pre-processed range Doppler and presence detection data through a configurable communication interface. In its default evaluation module (EVM) configuration, the radar transmits processed data via UART, which the ESP32-WROOM-32 can readily receive using one of its multiple hardware serial ports. The firmware initializes the UART at the specified baud rate (usually 921,600 bps for radar streaming) and implements a packet-parsing algorithm to decode the binary frame structure defined by TI's mmWave SDK. This interface simplifies integration during prototyping since UART requires only transmit, receive, and ground connections, making it ideal for early-stage validation and debugging using serial monitors or terminal emulators. Each data frame received is parsed for signal amplitude, range bin, Doppler shift, and phase variance, allowing the MCU to identify both large scale motion and micro-movements such as respiration. Temporal filtering techniques, such as moving average and low pass smoothing filters, are applied to suppress noise and minimize false detections caused by environmental reflections or electromagnetic interference.

While UART provides a simple, stable interface, it is limited in bandwidth for higher radar data rates or real-time streaming of multiple frame types. For this reason, future iterations of the system might transition radar communication to Serial Peripheral Interface (SPI), which is also supported by both the IWR6843AOP and the ESP32-WROOM-32. SPI provides significantly higher throughput, up to several megabits per second, enabling faster data transfers and more efficient synchronization between radar sampling and other sensor acquisitions. SPI's full-duplex capability would enable the MCU to simultaneously transmit configuration commands and receive radar data, thereby reducing latency and freeing up processing cycles for data fusion. Integrating the radar through SPI would further align it with the high-speed communication already used by the LCD display and potential UWB module, streamlining timing control and improving real-time performance under FreeRTOS task scheduling.
Radar configuration parameters, including chirp rate, frequency slope, frame duration, transmit power, and detection thresholds, are managed through initialization commands sent from the MCU to the radar over the chosen interface. The firmware schedules radar polling and data parsing within a dedicated FreeRTOS task, maintaining consistent frame timing independent of other sensor activities. The resulting radar output stream, which has been filtered and time-stamped, is then transmitted to the data fusion layer. Here, it undergoes cross-correlation with thermal and UWB signal responses to confirm the presence of a human being behind walls or rubble.

3.14.2. UWB Signal Processing (DWM3000)

The Decawave DWM3000 UWB module is integrated not for localization or ranging applications but as an RF sensing extension that enhances radar detection performance through dense or partially obstructed materials. The module communicates with the ESP32-WROOM-32 microcontroller using either the Serial Peripheral Interface (SPI) or Universal Asynchronous Receiver/Transmitter (UART) protocols, both of which are natively supported by the ESP32's flexible peripheral set. During early development, UART communication may be utilized for rapid prototyping due to its straightforward two-wire interface and simplified debugging via standard serial tools. UART allows the team to verify module functionality, transmit configuration commands, and view diagnostic data without extensive firmware overhead. However, for final integration and synchronized operation with the radar subsystem, SPI communication is the preferred interface. SPI offers substantially higher bandwidth and deterministic timing, allowing the microcontroller to retrieve channel impulse response and received signal strength indicator data at a much faster rate. This higher throughput enables near real-time processing by synchronizing UWB sampling intervals with radar frame timing, thereby enhancing data fusion and temporal correlation.

The DW3000 UWB transceiver communicates with the ESP32 over SPI (GPIO18 SCLK, GPIO23 MOSI, GPIO19 MISO, GPIO13 CS) and performs single-sided two-way ranging (SS-TWR) with the DWM3000EVB Pong responder on the Arduino Uno R4 WiFi anchor. Both TX and RX antenna delays are set to 16385, determined through iterative calibration to minimize ranging offset. The ranging state machine is protected by a FreeRTOS SPI mutex shared with the ILI9341 LCD. Distance measurements are fed into the 8-state Extended Kalman Filter as scalar X/Y/Z updates with 3.5σ innovation gating to reject multipath outliers. UWB confidence degrades quadratically as readings age beyond 1.5 seconds, with fusion alpha weighting of 0.70 for fresh readings, 0.50 for aging, and 0.25 for INS-only periods.

After filtering, the processed UWB signal data is normalized and time-stamped before being synchronized with radar outputs within the data fusion framework. This synchronization guarantees that both radar and UWB sensing modalities work together seamlessly, enhancing the system's reliability in verifying human presence through material barriers. By correlating energy fluctuations across UWB and radar data streams, the firmware can distinguish genuine human motion from static or environmental artifacts, significantly enhancing detection confidence. Future firmware updates may implement dynamic SPI clock scaling and DMA buffering to further optimize data throughput and minimize latency, enabling high-speed sensing without overloading the ESP32's processing cores.

3.14.3. Infrared Thermal Sensor (MLX90640)

The Melexis MLX90640 IR thermal array interfaces with the ESP32-WROOM-32 microcontroller via the I2C protocol, transmitting temperature data as a 32x24-pixel matrix. The I2C bus was chosen for this sensor because of its simplicity, low pin count, and compatibility with the ESP32's hardware I2C controllers. These controllers support clock speeds up to 1 MHz (Fast Mode Plus), which is more than enough to handle the MLX90640's maximum frame rate of approximately 16 Hz. The firmware configures the I2C interface with proper pull-up resistors and address mapping to ensure stable communication, particularly given the sensor's relatively high data payload per frame. During initialization, the microcontroller retrieves factory calibration constants and emissivity parameters stored in the sensor's onboard EEPROM, then applies Melexis's proprietary compensation algorithms to convert 16-bit raw pixel readings into precise temperature values in degrees Celsius. This compensation corrects for sensor offset drift, gain variation, and thermal gradients across the sensor die, which is critical for maintaining reliable human-heat detection accuracy in varying ambient conditions.

To maintain data integrity and mitigate frame-to-frame noise, the firmware implements a temporal averaging filter, which blends consecutive frames using a weighted moving average to stabilize pixel readings while preserving response time. A two-dimensional Gaussian smoothing kernel is then applied spatially across the thermal image. This process minimizes pixel level noise and sensor jitter while preserving sharp transitions at the boundaries of heat sources. These sharp transitions are crucial for distinguishing human-shaped signatures from background warmth. Each processed frame is analyzed for temperature gradients and intensity clustering to identify heat zones indicative of a human presence. A maximum intensity detection algorithm scans the filtered frame for temperature anomalies that exceed a dynamic threshold above the ambient baseline, adapting automatically to environmental conditions.

The ESP32 firmware manages this process through a dedicated I2C polling task running within the FreeRTOS environment, ensuring the sensor is sampled at consistent intervals synchronized with the radar subsystem's frame rate. This synchronization allows direct temporal alignment between radar detections of motion or breathing and corresponding thermal readings of body heat, significantly increasing the reliability of life detection outcomes. To prevent data collisions on the shared I²C bus, the firmware schedules sequential read operations with prioritized timing slots across all connected devices: the MLX90640 thermal camera, MPU9250 IMU, BMP280 barometric sensor, and CAT9555 GPIO expander. Error handling routines detect communication interruptions and automatically reinitialize the I2C connection if a bus hang or NACK condition occurs, preserving long term operational stability.
Finally, the processed thermal data, which includes the centroid coordinates of the warmest regions, regional temperature averages, and maximum temperature gradients, is normalized and transferred to the data fusion subsystem. There, it is

time stamped and cross correlated with radar and UWB sensor outputs. This multi-sensor integration allows the firmware to confirm the presence of a heat-emitting body in conjunction with motion or micro-vibration data, dramatically reducing the probability of false positives. Future firmware enhancements may incorporate adaptive temporal filters that adjust averaging strength based on motion activity or ambient temperature changes, improving response precision in dynamic search-and-rescue conditions.

3.14.4. Inertial Navigation System

The custom INS stack integrates an MPU9250 9-axis IMU (accelerometer, gyroscope, and magnetometer) and a BMP280 barometric pressure sensor, both communicating with the ESP32-WROOM-32E over the shared I²C bus (GPIO21 SDA, GPIO22 SCL) alongside the MLX90640 thermal camera and CAT9555 GPIO expander. The I²C interface was chosen for its ability to multiplex multiple sensors on a common two-wire connection, minimizing pin usage while maintaining sufficient communication speed for all connected devices.

During initialization, quick calibration routines collect 300 samples for accelerometer and gyroscope bias estimation and 12 samples for barometric altitude zeroing. These routines execute at power-up and again each time the operator presses the START button, which anchors the local origin and resets the dead-reckoning state. The MPU9250 supplies high-rate acceleration and angular velocity data that are fused using the Madgwick orientation filter, a gradient-descent-based AHRS algorithm that produces a quaternion orientation estimate from accelerometer and gyroscope inputs without requiring a magnetometer, making it robust to the magnetic disturbances common in cluttered indoor environments. Yaw is maintained separately by integrating raw gyro Z-axis output minus a calibrated bias, rather than relying on the Madgwick yaw output, which drifts too quickly for indoor navigation. The magnetometer provides a tilt-compensated compass heading used as a fallback and for radar map orientation when valid.

Linear acceleration is rotated into the world frame using a 3×3 rotation matrix derived from roll, pitch, and yaw, then integrated with velocity damping and a still-detection gate to produce dead-reckoning position estimates. The still-detection gate requires a configurable number of consecutive frames where total acceleration magnitude is within a narrow band of 1 g and gyroscope magnitude is below threshold before zeroing velocity, preventing drift accumulation during standstill. Short-term tracking drift is bounded by fusing UWB range measurements through an 8-state Extended Kalman Filter incorporating 3D position, 3D velocity, yaw heading, and a barometer
bias term.

The BMP280 measures ambient pressure converted to altitude via the hypsometric formula, supplying the Z-axis correction input to the EKF alongside UWB ranging. The BMP280 temperature output is additionally

fused into the MLX90640 thermal classification pipeline, scaling the background model adaptation rate with ambient conditions to reduce false positives from heated surfaces in warm deployment environments. Together, the MPU9250 and BMP280 provide the orientation, displacement, altitude, and ambient reference data needed to contextualize radar and thermal sensor readings, compensate for operator-induced motion, and place victim detections in a spatial coordinate frame for transmission to the companion iOS and Android applications over BLE.

3.14.5. User Interface and Feedback

The UI firmware serves as the operator's primary point of interaction with the 2HADRAD system, enabling visual, tactile, and auditory feedback under varying environmental conditions. The UI comprises a 3.2-inch ILI9341 TFT LCD display, three OSRAM R976 red indicator LEDs, three APEM 1ZW0913611806 pushbuttons, and a 3 W, 4 Ω speaker driven by a MAX98357A digital audio amplifier. Together, these components ensure clear and reliable signaling in low-visibility or high-noise rescue environments. The firmware running on the ESP32-WROOM-32coordinates all interface elements using a combination of SPI, I²S, and GPIO communication channels, allowing independent yet synchronized control across subsystems.

The ILI9341-based TFT LCD module communicates with the MCU through the SPI bus, selected for its efficient data transfer and compatibility with the ESP32 hardware SPI peripheral. The SPI interface allows the system to update graphical elements such as sensor readings, detection indicators, and system status information with minimal processing overhead. During initialization, the firmware configures the display controller, including orientation, pixel format, and refresh parameters, before loading graphical assets and interface layouts. A dedicated display task manages periodic screen updates while prioritizing critical notifications such as confirmed detections or system warnings. This task structure ensures that the display remains responsive without interrupting sensor processing or other real-time system functions.

The status LED is connected to GPIO4, configured as a digital output and driven HIGH to illuminate when a high-confidence human detection occurs. The firmware asserts the LED on the same edge as the detection alert, providing at-a-glance confirmation without PWM or timer-based blinking patterns in the current implementation.

User input is provided by a 5-way navigation button whose signals are read through the CAT9555 I²C GPIO expander rather than direct ESP32 GPIO pins. The four cardinal directions navigate the on-screen menu, and the center button selects actions, covering all operator functions including starting and stopping scans, marking victim positions, switching display modes, and initiating BLE connection. Button state is read by polling the CAT9555 port register on each loop iteration; a software debounce check using millisecond-level timing suppresses false triggers from vibration or electrical noise. Button events

are processed in the main loop rather than via interrupt service routines, consistent with the cooperative scheduling model used across all UI-related tasks. A toggle switch on the enclosure handles power, separate from the navigation button, so the operator cannot accidentally power down the device during field operation.

Auditory feedback is provided through a MAX98357A digital Class-D audio amplifier paired with a 3 W, 4 Ω speaker. The amplifier receives audio data directly from the ESP32 using the I²S digital audio interface, allowing the system to output prerecorded alert tones without requiring an external digital-to-analog converter. A dedicated audio task handles playback of audio files stored in program memory and transmits the data stream to the amplifier via the I²S peripheral. This configuration enables the system to generate clear audible alerts that remain audible in noisy rescue environments while maintaining high power efficiency.

All UI firmware tasks operate under FreeRTOS, enabling concurrent operation with radar and thermal sensing tasks while maintaining real-time responsiveness. Task prioritization ensures that critical user interactions and detection alerts are processed immediately without disrupting sensor data acquisition. The firmware also includes failsafe notification mechanisms, such as distinctive LED patterns or audible alerts triggered during system faults or communication failures. These safeguards ensure that operators receive continuous feedback regarding system status even if one sensory channel becomes ineffective due to environmental conditions.

Collectively, the UI firmware integrates all user-facing components into a cohesive and low-latency control interface that enhances operational awareness and reliability. By combining SPI communication for display control, I²S audio output for alert playback, and interrupt-driven GPIO handling for user inputs, the 2HADRAD system maintains intuitive operation and dependable feedback in demanding field environments.

3.14.6. Data Fusion and Algorithm

At the core of the 2HADRAD’s processing architecture lies the data fusion firmware, which synthesizes inputs from the radar, UWB, thermal, and INS subsystems into a single, coherent life detection output. INS orientation estimation uses the Madgwick gradient-descent AHRS filter with separate gyro-Z yaw integration; position fusion is performed by the 8-state EKF described in Section 7.1.10. Each sensor operates at a unique sampling rate and communication bandwidth. To ensure consistent output, the ESP32-WROOM-32’s firmware employs time stamping, normalization, and frame alignment procedures. Every data stream, such as radar Doppler data, UWB channel impulse response, thermal frame, or INS motion vector, is assigned a synchronized timestamp. This timestamp is derived from the MCU’s high-precision hardware timer. These timestamps allow the fusion algorithm to

correlate simultaneous signal events across different modalities, compensating for latency differences between sensor acquisition intervals.

The fusion routine runs as a dedicated FreeRTOS task with the highest processing priority, ensuring uninterrupted execution even under heavy peripheral activity. Data is shared between tasks through thread-safe message queues, preventing frame loss or overlap during concurrent sensor operations. The firmware performs multi-sensor correlation, weighting radar and UWB data more heavily for motion-based detection while using thermal imagery for biological confirmation. Radar and UWB outputs are first evaluated through cross correlation functions to determine temporal and amplitude overlap in their reflected signal profiles. A multi-threshold decision logic is then applied, requiring both sustained radar motion signatures and corresponding UWB energy peaks that persist within a defined time window. Once this temporal condition is satisfied, the algorithm references the most recent thermal frame to confirm the presence of a localized heat source above ambient temperature threshold. The INS data is simultaneously processed to characterize device orientation and motion. By analyzing acceleration and angular velocity vectors, the firmware identifies user-induced movement or tilt that could generate false radar or UWB signals. When motion beyond a defined threshold is detected, the fusion algorithm temporarily suppresses life-detection classification until the device stabilizes. This approach guarantees that environmental vibrations or operator repositioning won't cause false positives, thereby maintaining a high level of detection accuracy in the field.

Filtering and decision making occur in stages: radar and UWB data undergo temporal smoothing and energy envelope extraction, thermal data is analyzed using intensity clustering, and INS data is filtered through a complementary filter to isolate genuine device motion. The resulting features are fed into a rule-based decision matrix, which classifies detections as either "transient," "suspected," or "confirmed." The firmware elevates the classification to "life detected" only when multiple modalities consistently reinforce each other over several frames. For instance, persistent radar oscillations synchronized with UWB amplitude fluctuations, and a stable thermal hotspot are required. Once a detection is confirmed, the fusion output is routed to the UI subsystem, where both visual and auditory alerts are triggered in real time. The LCD displays an alert banner and highlights detection confidence levels, while an LED, MAX98357A amplifier and the speaker provide redundant cues for high-noise environments. The firmware records each detection event with timestamp, confidence score, and sensor contribution weight in non-volatile memory, allowing later review or algorithm refinement.

Looking ahead, the rule based fusion framework is intentionally modular to support transition toward a machine-learning-based classifier. Future iterations will incorporate lightweight embedded AI models, such as decision trees or neural networks, that are trained on multi-sensor datasets collected during field

testing. These models dynamically adjust detection thresholds based on environmental context and noise patterns, enabling adaptive fusion intelligence that can identify subtle human signatures in complex and cluttered conditions. Together, the data fusion firmware transforms the raw, asynchronous outputs of individual sensors into a unified, reliable life-detection decision engine. By combining temporal synchronization, sensor-specific filtering, and correlation-based validation, it provides the computational foundation for the 2HADRAD's mission: rapid, accurate, and field-ready human presence detection through obstacles and debris.

3.14.7. Communication Summary

Each subsystem's firmware is developed in C/C++ using either the Arduino framework or the Espressif IoT Development Framework (ESP-IDF), which are the two primary environments supported by the ESP32-WROOM-32. The Arduino environment offers a simplified interface and extensive library support, making it an excellent choice for rapid prototyping and hardware validation. On the other hand, the ESP-IDF provides advanced real-time task control, memory optimization, and low-level peripheral access through FreeRTOS, enabling the development of robust, production-grade firmware. The choice between these environments provides flexibility for team members to work within their preferred ecosystem while ensuring interoperability and code consistency across modules. To ensure efficient and reliable communication between the MCU and all connected subsystems, optimal data protocols were selected based on bandwidth, timing, and synchronization requirements. The IWR6843AOP radar module, which produces large volumes of motion and Doppler data, would benefit most from SPI communication due to its high throughput and low latency, ensuring fast frame transfers and minimal signal delay during real-time detection. However, since the radar's evaluation module (EVM) is configured by default for UART output, the firmware will initially use UART communication for simplicity. Later design revisions will transition to SPI for improved performance. The DWM3000 UWB module can also operate over SPI or UART. However, SPI is preferred in the final design to maintain synchronization with radar sampling intervals and minimize overhead during data bursts. The MLX90640 IR thermal array and INS module both operate optimally over I2C, which supports multiple peripheral devices on a shared bus while maintaining sufficient speed for moderate-rate sensor polling. Finally, GPIO and PWM interfaces are reserved for UI components such as LED, 5-way navigation button, the amplifier, and the speaker, where deterministic control and minimal latency are required for real-time operator feedback.

Regardless of the selected environment, the firmware architecture ensures consistent timing, communication reliability, and modular expandability. Filtering algorithms, such as temporal averaging, Gaussian smoothing, and complementary orientation estimation, ensure signal fidelity even in noisy operating conditions. Collectively, these firmware modules constitute a flexible and scalable software backbone for the 2HADRAD system, enabling future

enhancements like BLE connectivity, machine-learning-based detection, and advanced data visualization capabilities.

3.15. Communication Frameworks for Smartphone App Development

When designing a smartphone-connected embedded device, there are a handful of options for which communication framework to use. The choice affects power consumption, responsiveness, user experience, and scalability.

3.15.1. Bluetooth Low Energy (BLE)

BLE is part of the Bluetooth 4.0+ specification and is specifically tuned for low-power, short-range communication. It's designed around brief data exchanges rather than continuous streams, which makes it ideal for mobile-connected sensing devices, health monitors, and wearables, which is precisely our design case. The ESP32 family (WROOM-32, S3, C6) supports BLE peripheral mode, enabling our device to advertise itself to smartphone apps.
BLE communication has some drawbacks, including low data throughput (theoretically around 1-2 Mbps, usually 200-300 kbps) and a limited range. However, due to the nature of our data, not much throughput is required, and our use-case is for the smartphone to be relatively close to the device (no more than 20 meters).
BLE aligns well because we want real-time updates with minimal power, the ESP32 line implements complete BLE GATT APIs, and both OS ecosystems handle BLE securely without internet.

3.15.2. WiFi (HTTP / WebSocket)

Wi-Fi connectivity leverages standard TCP/IP for data transfer. Our ESP32 can act as either a station (connected to a router or access point), or a soft access point (AP) (hosting its own network). Once on the same network, the MCU and smartphone communicate via HTTP REST endpoints (pull-based or push via POST) and WebSockets (persistent real-time TCP connection). This model suits dashboards, configuration pages, or cloud-connected scenarios.
Benefits of this setup include high bandwidth, compatibility with different MCUs, and much higher distance than BLE. However, some drawbacks include a higher power consumption, setup friction (user must join network and manage credentials), and potential latency issues.

Wi-Fi is well-suited for stationary setups (such as a lab or a smart home product) with power availability, cases where we'd want to use a web dashboard or local API access over the LAN, or scenarios where multiple mobile clients connect simultaneously. These are not really scenarios of use for our device, which is handheld, prefers minimal setup friction, and doesn't need any APIs.

3.15.3. MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight publish/subscribe protocol built over TCP/IP, engineered for IoT systems that span devices and networks. Rather than a smartphone connecting directly to the MCU, both connect to a broker (cloud or local server). Messages consist of simple topics (e.g., device123/co2, device123/heartbeat). This decouples data producers (MCUs) and consumers (apps) entirely, enabling robust asynchronous systems.

Some benefits of MQTT include its scalability to multiple devices and users, power efficiency, and cloud integration. Some drawbacks are that it requires an internet connection and a broker setup, it is less less latent than BLE, it requires cloud management, and it increases firmware complexity.
MQTT is excellent if we envision our device evolving toward a fleet of cloud-connected monitoring (multiple deployed units), real-time data dashboards from anywhere, and multi-user collaboration or server-side analytics. However, for localized personal sensing, direct BLE or Wi-Fi is much simpler and fit better into the scope of our project, so we are not going with MQTT.

3.15.4. USB (Wired Communication)

USB is the traditional physical data link between embedded devices and host computers or smartphones. Only some MCUs (like ESP32-S3) natively support USB 2.0 OTG, enabling them to act either as a serial device or even HID (Human Interface Device)/MSC (Mass Storage Class) device. With appropriate adapters, such as a USB-C to micro-USB cable, smartphones can directly interface for data transfer or configuration using native serial drivers.

Benefits of this method of communication include reliability, high throughput, security, instant pairing and setup. Some drawbacks include the fact that it requires physical tethering, and that it adds unnecessary mechanical complexity. USB is most useful for development and debugging purposes, such as serial logging and fast firmware updates. It’s also suitable for calibration or maintenance connections where a stable wired link is crucial. However, it’s not a viable option as the primary UI in portable or consumer devices. Since USB is not suitable for our device usage conditions, such as the inconvenience of always being connected to a smartphone, we opted not to use this method for smartphone connection.

Communication Protocol	Typical Use Case	Pros	Cons
BLE	Direct local link; common in ESP32/S3/C6	Power-efficient, works offline	Lower bandwidth; requires phone proximity

Communication Protocol	Typical Use Case	Pros	Cons
Wi-Fi (HTTP / WebSocket)	Local LAN or MCU AP mode	Simple TCP/IP; easy browser integration	MCU must run local web server; more power use
MQTT over Internet	Cloud integration	Reliable, standard IoT protocol; good for multi-device systems	Requires broker setup; needs network connectivity
USB (if MCU supports it like ESP32-S3)	Debugging, wired link	Fast, direct	Limited user convenience, adds physical complexity

Table 25: Smart Phone Application Communication Comparison

3.16. Smartphone Application Development

3.16.1. Development Infrastructure Comparison

An advanced goal of our project is to develop a smartphone app to display output data from the device, which is data the device MCU receives from the sensors (radar, IR, UWB, etc). For app development, there are two main options: Android or iOS.
Some factors that determine whether to develop for iOS or Android include ease of development, integration with the MCU low setup friction, and support for data visualization (UI). Due to such factors, we believe native Android development is the most pragmatic path and not iOS. Android provides flexible access to Bluetooth, Wi-Fi, and USB APIs. We can run the APK directly on any Android phone or even an emulator, and there will be no provisioning or Apple developer account friction. Within the Android development space, there are a handful of approaches to take, varying in programming languages, UI implementation, novelty, community support and documentation, et cetera.

3.16.2. Android Technology Comparison

3.16.2.1. Android Native (Kotlin + Jetpack Compose)

This is Google’s official modern pathway for Android. We would use Kotlin, with Android Studio. UI will be built using Jetpack Compose, which is a declarative toolkit like React for native components.
Benefits of this approach are a modern, clean syntax and reduced boilerplate versus Java, excellent Android documentation and tooling. Jetpack Compose simplifies the rapid iteration of UIs for dashboards and gauges, which is essentially our objective. This works seamlessly with any ESP32 BLE example. In addition, this approach gives us full access to system permissions, background services, and notifications. One drawback of this approach is its relatively steep learning curve, as we are new to Kotlin. Jetpack Compose is still

evolving, which means some advanced UI behaviors are not well-documented. Additionally, Android Studio can be resource-intensive on modest computers. Overall, this is our choice because it's modern and thus relevant to apps built today, well supported documentation, good UI tools, and total freedom in terms of the customization of our app.

3.16.2.2. Android Native (Java + XML Layouts)

This approach is the 'classic' Android stack (Java + XML). It is still fully supported and many BLE and HTTP tutorials exist in Java, including Espressif's legacy Android templates.
Benefits of this approach include the fact that it has huge existing codebase examples, it offers easier adoption for anyone with experience in Java or classic OOP patterns, and it is stable as almost any Android version supports it.
Drawbacks of this approach include the fact that it has verbose syntax and more boilerplate than Kotlin/Compose. Its UI iteration is slower as it must recompile layout XML, it is an older approach and thus less future-proof.
We wanted to go for a more modern development approach and avoid Java, and the UI tools in this approach seem less attractive and flexible. Hence, we did not choose this development path.

3.16.2.3. Android + MIT App Inventor / Kodular (Visual Builder)

MIT App Inventor, Kodular, and similar web-based tools allow graphical block programming. Devs drag/drop UI components (buttons, labels, charts) and assign logic visually. The benefits of this approach are that it boasts a very fast development cycle, no deep programming expertise required. It's great for visual demonstrations or classroom prototypes and offers a free platform (MIT App Inventor) with built-in emulation.
One downside of this approach is that it offers limited customization beyond what blocks allow, which may result in a less professional-looking GUI compared to native approaches. In addition, it cannot implement complex background threads or advanced parsing easily, and the BLE stack is less reliable on recent Android versions. This approach is not our choice due to limited UI customization and a non-traditional development pattern.

3.16.2.4. Android with Flutter (Dart)

Flutter is a high-level way to build Android apps in Dart (a programming language developed by Google). It offers UI deployed via native rendering. In terms of complexity, this approach sits between native based implementation (building platform-specific apps using each platform's official languages and tools) and web-based implementation (building apps that run in a web-based application, such as HTML/CSS/JavaScript frameworks).
Benefits of this technology are that it offers a fast reload and instant preview of UI changes, structured UI that is visually impressive, and easier charting and data visualization using libraries. Drawbacks include the fact that there's extra start-up overhead (installing Flutter SDK), and it's not as low-level as native Android if we

need fine control. We did not go with this choice because the learning curve of Flutter and Dart seems too steep for the relatively simple application we are trying to develop.

3.16.2.5. Android Native (Java + XML Layouts)

This approach is the ‘classic’ Android stack (Java + XML). It is still fully supported and many BLE and HTTP tutorials exist in Java, including Espressif’s legacy Android templates.
Benefits of this approach include the fact that it has huge existing codebase examples, it offers easier adoption for anyone with experience in Java or classic OOP patterns, and it is stable as almost any Android version supports it.
Drawbacks of this approach include the fact that it has verbose syntax and more boilerplate than Kotlin/Compose. Its UI iteration is slower as it must recompile layout XML, it is an older approach and thus less future-proof. We wanted to go for a more modern development approach and avoid Java, and the UI tools in this approach seem less attractive and flexible. Hence, we did not choose this development path.

3.16.2.6. Android + MIT App Inventor / Kodular (Visual Builder)

MIT App Inventor, Kodular, and similar web-based tools allow graphical block programming. Devs drag/drop UI components (buttons, labels, charts) and assign logic visually. The benefits of this approach are that it boasts a very fast development cycle, no deep programming expertise required. It’s great for visual demonstrations or classroom prototypes and offers a free platform (MIT App Inventor) with built-in emulation.
One downside of this approach is that it offers limited customization beyond what blocks allow, which may result in a less professional-looking GUI compared to native approaches. In addition, it cannot implement complex background threads or advanced parsing easily, and the BLE stack is less reliable on recent Android versions. This approach is not our choice due to limited UI customization and a non-traditional development pattern.

3.16.2.7. Android with Flutter (Dart)

Flutter is a high-level way to build Android apps in Dart (a programming language developed by Google). It offers UI deployed via native rendering. In terms of complexity, this approach sits between native based implementation (building platform-specific apps using each platform’s official languages and tools) and web-based implementation (building apps that run in a web-based application, such as HTML/CSS/JavaScript frameworks).
Benefits of this technology are that it offers a fast reload and instant preview of UI changes, structured UI that is visually impressive, and easier charting and data visualization using libraries. Drawbacks include the fact that there’s extra start-up overhead (installing Flutter SDK), and it’s not as low-level as native Android if we need fine control. We did not go with this choice because the learning curve of

Flutter and Dart seems too steep for the relatively simple application we are trying to develop.

Approach	Language / Framework	Communication Libraries / SDKs	UI Development Style	Best For
Android Native (Kotlin + Jetpack Compose)	Kotlin (Android Studio)	Android BluetoothGatt, Retrofit (HTTP), MQTT Client	Modern declarative (Compose)	Clean modern prototype, BLE based MCU comms-based MCU comms
Android Native (Java + XML Layouts)	Java (Android Studio)	Same as above + legacy libs (OkHttp, Paho MQTT)	XML & View system	Stable legacy base, extensive examples
Android + MIT App Inventor / Kodular	Block based visual programming-based visual programming	Built-in BLE and Web APIs-in BLE and Web APIs	Drag and drop GUI-and-drop GUI	Ultra-fast demos, limited customization
Android with Flutter (Dart)	Dart (Flutter SDK)	flutter_blue_plus (BLE), http/mqtt_client	Declarative widget tree	Quick modern UI with BLE, single codebase optional

Table 26: Android Technology Comparison

CHAPTER 4

4.1. Introduction to Standards and Design Constraints

This chapter defines the rules and limits that govern the design of the 2HADRAD device. Since the system integrates multiple radio technologies, 60 GHz FMCW radar, 2.4 GHz BLE, and UWB ranging, it can't be treated as a simple embedded project; it must comply with U.S. FCC Part 15 requirements for both intentional and unintentional radiators to operate legally in unlicensed spectrum and to avoid causing interference to other services. At the same time, several industry and international standards (IEEE 802.15.4a/z for UWB, GNSS ICDs for multi-constellation positioning, and I²C/SPI serial bus specifications) dictate how subsystems communicate, how signals must be structured, and what levels of accuracy or interoperability are expected. Beyond standards, the device is also constrained by practical engineering factors, available power (≤ 10 W), passive

thermal dissipation inside a small enclosure, handheld mechanical form factor, limited computational resources on the ESP32, and the cost ceiling typical of a university prototype. Finally, the project is constrained by ethical and safety considerations: RF exposure must remain within established limits, collected data must remain on-device, and the intended use must align with life-detection and rescue scenarios. The sections that follow translate these regulatory, technical, and practical requirements into concrete design decisions for RF emissions, communication buses, enclosure layout, power distribution, and verification/documentation practices.

4.1.1. FCC Part 15 – Radio Frequency Devices

2HADRAD both transmits and receives radio frequency (RF) energy through its 60 GHz radar module, and it communicates with external devices using BLE. Therefore, our device is classified by the U.S. Federal Communications Commission (FCC) as an intentional radiator under Title 47 CFR Part 15, which means it is a “device that intentionally generates and emits radio frequency energy by radiation or induction that may be operated without an individual license,” [1]. Thus, our team must ensure that we adhere to the standards for intentional radiators. These standards include strict limits on output power, spectral emissions, and spurious radiation to prevent interference with licensed communication services.
Furthermore, under Part 15 Subpart B, all digital electronics within the device, such as the microcontroller, analog-to-digital converters, and switching regulators, etc., must comply with radiated and conducted emission limits for unintentional radiators, which the FCC defines as “a device that by design uses digital logic, or electrical signals operating at radio frequencies for use within the product, or sends radio frequency signals by conduction to associated equipment via connecting wiring, but is not intended to emit RF energy wirelessly by radiation or induction.” To meet these requirements, the PCB layout will use continuous ground planes, short return paths, and decoupling capacitors on every supply rail. The switching regulators are enclosed within grounded copper pours, and all high-speed clock traces are impedance-matched to reduce harmonic radiation.

The BLE radio, operating in the 2.4 GHz ISM band, is regulated by Part 15 Subpart C, which regulated low-power intentional radiators. The module transmits below +10 dBm EIRP, occupies the 2400-2483.5 MHz band, and employs frequency-hopping to minimize interference. The microcontroller, the ESP32-WROOM, is already FCC-certified, so integration only requires maintaining its approved antenna configuration and keeping trace lengths short to maintain pre-designed radiated-emission characteristics.
Additionally, the 60 GHz radar module, the TI IWR6843AOP EVM, operates under Part 15 Subpart C, Section 15.255, which governs unlicensed use of the 57–71 GHz ISM band. This section specifies the maximum allowable equivalent isotropic radiated power (EIRP), power density, and out-of-band emissions to prevent interference with licensed millimeter-wave services such as point-to-point

backhaul and satellite downlinks. The rule limits continuous-wave emissions to +40 dBm EIRP peak and requires that any emissions outside the 57–71 GHz range be attenuated by at least 20 dB below the fundamental. Our radar's antenna-on-package (AoP) design confines energy within this band, and the measured EIRP is well below the regulatory maximum, allowing for full compliance without additional attenuation circuitry.
Through these design measures, the 2HADRAD is compliant with the FCC Part 15 Subparts B and C, covering both intentional and unintentional radiators, thereby ensuring lawful operation in unlicensed bands and preventing harmful interference to other users of the electromagnetic spectrum.

4.1.2. IEEE 802.15.4 UWB Standard (802.15.4a / 802.15.4z)

The 2HADRAD system utilizes an UWB transceiver, the DWM3000EVB, that operates under the IEEE 802.15.4a/z physical layer standard for high-precision ranging and localization. The IEEE 802.15.4 framework defines low-rate wireless personal area networks (LR-WPANs), but its UWB extensions (802.15.4a and 802.15.4z) specify impulse-radio transmission techniques optimized for ToF measurements and secure ranging. These standards are crucial to our project because they establish how short-duration pulses are generated, modulated, and time-synchronized to allow for centimeter-level accuracy while not interfering with other wireless systems in the 3.1–10.6 GHz spectrum.

Under IEEE 802.15.4a, UWB signals are transmitted as nanosecond-scale pulses spread across a wide bandwidth, typically exceeding 500 MHz, to achieve extremely fine temporal resolution. This spread-spectrum approach produces a low power spectral density (below –41.3 dBm/MHz), ensuring that UWB transmissions do not cause harmful interference to narrowband systems such as Wi-Fi, Bluetooth, or LTE. The 802.15.4z revision, introduced later, enhanced security and improved ranging robustness by defining Secure Ranging Exchange (SRE) protocols, Scrambled Timestamp Sequences (STS), and improved pulse-position modulation schemes. These updates allow UWB modules like the DWM3000EVB to authenticate distance measurements and resist spoofing or relay attacks, which is an important feature when multiple devices are operating in proximity during search-and-rescue operations.

In the 2HADRAD design, the UWB module supports ToF ranging for the handheld device. The system operates within the 6.5–8.0 GHz band, fully aligned with FCC Part 15 Subpart F (for UWB transmitters) and the IEEE spectral mask requirements. Transmission duty cycles, burst durations, and pulse repetition frequencies are configured to maintain compliance with both the average and peak power limits defined in the standard. The DWM3000EVB's hardware implementation employs time-hopping and preamble codes specified in the 802.15.4a/z PHY to minimize cross-interference and support multi-user operation.

The UWB's PCB incorporates controlled impedance microstrip lines for the antenna feed and maintains a 50 Ω characteristic impedance across the RF path. Ground-via stitching surrounds the antenna region to confine the radiation field and reduce coupling into adjacent circuits.
Through adherence to these standards and verification practices, the 2HADRAD ensures proper UWB communication in accordance with both IEEE 802.15.4a/z and FCC Part 15 Subpart F.

4.1.3. The 2HADRAD system GNSS System Standards (GPS compliance)

The 2HADRAD system incorporates GNSS as an auxiliary absolute positioning source, parsed via the TinyGPS++ library over a time-multiplexed UART shared with the radar configuration interface. When a valid fix is available, the system produces position coordinates compliant with the output formats defined by the major constellation ICDs: IS-GPS-200 (GPS L1 C/A), the Galileo OS SIS ICD, BeiDou B1I ICD, and GLONASS ICD. The TinyGPS++ library decodes NMEA 0183 sentences — specifically GGA and RMC — which carry position, velocity, and time data structured according to these constellation standards. The system therefore inherits ICD compliance through the GNSS module's onboard receiver firmware rather than implementing signal-level decoding directly on the ESP32.

Performance targets for the GNSS subsystem align with the thresholds defined in RTCA DO-229D and IEC 61108-1, including a horizontal position accuracy of approximately 1.5 m CEP under open-sky conditions and a time-to-first-fix consistent with cold-start specifications for multi-constellation receivers. The system operates on the L1/E1 band (≈ 1575 MHz) in accordance with ITU-R Recommendation M.1477 governing spectral characteristics for civil GNSS reception.

In practice, GNSS is treated as an opportunistic input. Indoors or under debris — the primary deployment environment for 2HADRAD — GNSS signal is typically unavailable and the system falls back to UWB-corrected inertial dead-reckoning via the 8-state Extended Kalman Filter. When a valid fix is reacquired, absolute coordinates are incorporated into the fusion layer and transmitted with victim detection records to the companion iOS and Android applications over BLE. This hybrid approach satisfies the positioning accuracy requirements of the design while remaining robust to the signal environments encountered in search-and-rescue operations.

4.1.4. I2C / SPI Interface Standards (Serial Bus Communication)

The 2HADRAD system employs industry-standard serial communication protocols, Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI), to ensure reliable and standardized data exchange between the microcontroller and its peripheral sensors. The I²C bus, originally defined by Philips (now NXP), provides a two-wire communication interface consisting of a data (SDA) and clock (SCL) line shared among multiple devices. It supports multi-master, multi-slave configurations and uses open drain signaling with pull-up resistors to maintain proper logic levels. Within 2HADRAD, the I2C interface operates at 3.3 V in Fast Mode (400 kHz) to connect low-bandwidth devices such as the barometric sensor (BMP280), magnetometer (BMM150), and IR thermal sensor. Trace lengths and pull-up resistor values are selected in accordance with the I2C specification to minimize capacitive loading and preserve waveform integrity.

For higher-speed data exchange, the system utilizes the SPI standard, first established by Motorola, which defines a four-wire interface consisting of a master clock (SCLK), chip select (CS), master-out–slave-in (MOSI), and master-in–slave-out (MISO) lines. SPI enables full-duplex communication at several megahertz and is used for bandwidth-intensive modules such as the UWB transceiver and radar interface. Following the SPI electrical and timing requirements, all high-speed lines are impedance-controlled and routed with short, direct connections to reduce crosstalk and electromagnetic interference. By adhering to these standardized serial bus protocols, the 2HADRAD ensures interoperability, data consistency, and timing synchronization between all sensing subsystems, supporting robust and deterministic communication across the device.

4.1.5. IEC Safety and EMC Standards (ESD, EMI mitigation)

The 2HADRAD system is designed in accordance with International Electrotechnical Commission (IEC) safety and electromagnetic compatibility (EMC) standards to ensure reliable and interference-free operation in mixed electronic environments. Specifically, it aligns with IEC 61000-4 series for EMC immunity testing and IEC 61326 for electrical equipment operating in industrial and scientific environments. These standards define procedures and performance limits for exposure to electrostatic discharge (ESD), electromagnetic interference (EMI), and radiated or conducted disturbances, ensuring that the device maintains normal function without causing or suffering from disruptive emissions.

To comply with IEC 61000-4-2 (ESD immunity), the design incorporates transient voltage suppression diodes on external I/O lines, grounded shielding, and a copper pour around connectors and exposed interfaces. These measures protect sensitive circuits such as the radar front end and GNSS receiver from

electrostatic discharges up to ±8 kV (contact) and ±15 kV (air), meeting the IEC test levels for portable electronics. For EMI mitigation under IEC 61000-4-3 and -4-6, the PCB layout uses continuous ground planes, ferrite bead filtering on power inputs, and differential signal routing for high-speed lines like SPI and UART. The radar module is further enclosed within a metal EMI shield to contain radiated emissions at 60 GHz and prevent coupling into digital subsystems. By following these IEC safety and EMC practices, the 2HADRAD minimizes susceptibility to electrical disturbances, prevents interference with nearby equipment, and ensures user safety and operational stability across diverse electromagnetic environments.

4.1.6. ISO / IEC Quality and Testing Standards

While the 2HADRAD prototype will not undergo formal certification or laboratory testing, its design and development process reference key ISO and IEC quality and testing standards to promote consistency, reliability, and traceability. Standards such as ISO 9001:2015 for quality management and IEC 60068 for environmental durability provide a framework that guides the team's approach to documentation, component selection, and design verification. Although full qualification testing will not be performed, the principles of these standards, structured reviews, defined acceptance criteria, and iterative validation, are applied throughout the development cycle to maintain engineering discipline and repeatability.

For sensor calibration, the team follows best practices derived from ISO/IEC 17025, which defines general requirements for testing and calibration accuracy. Instead of formal laboratory procedures, sensors such as the barometer, magnetometer, and IMU will be evaluated through controlled functional tests and manufacturer datasheet validation to ensure consistent readings under typical operating conditions.

By referencing these ISO and IEC frameworks conceptually rather than implementing full compliance testing, the 2HADRAD project maintains a realistic balance between academic prototyping and professional engineering standards, ensuring design credibility, organized workflow, and defensible technical documentation without exceeding the scope of a student research prototype.

4.1.7. Power Constraints

The 2HADRAD is a portable, battery-powered life detection and localization device designed for use in field environments where access to external power is limited. Therefore, strict adherence to power efficiency and distribution constraints is critical to maintain system performance, reliability, and operating time. The overall system is designed to operate within a 10 W total power budget, which encompasses all sensing modules, the microcontroller, display, and UI components. To meet this requirement, each subsystem has been selected and configured to balance performance with low power consumption

while maintaining sufficient processing speed and sensing accuracy for real-time operation.

The most power-intensive component in the system is the TI IWR6843AOP radar module, which consumes approximately 2.0–2.5 W during continuous operation. This radar serves as the primary sensing element, performing frequency-modulated continuous wave (FMCW) transmissions and on-chip digital signal processing. Its internal design integrates both analog front-end and baseband processors, minimizing the need for external computation but necessitating dedicated power regulation and thermal management. To ensure stable operation, the radar is supplied through a dedicated 3.3 V DC-DC buck converter with input filtering capacitors and ferrite beads to suppress switching noise that could otherwise interfere with nearby sensors.

Supporting sensors including the MPU9250 9-axis IMU, BMP280 barometric pressure sensor, and CAT9555 GPIO expander together contribute approximately 0.13 W combined at 3.3 V during continuous operation. These low-power devices operate intermittently, polling data at controlled intervals rather than continuously, reducing average consumption. The DWM3000EVB UWB module consumes approximately 100–150 mW during active ranging, and its duty cycle is managed through firmware scheduling to avoid simultaneous high-current draws from multiple RF subsystems. Similarly, the ESP32-WROOM microcontroller, which handles data fusion, display control, and wireless communication, operates around 500–600 mW under dual-core processing, with deep-sleep modes utilized when radar or ranging tasks are inactive.

The system’s power distribution architecture employs switching regulators to provide isolated rails for analog, digital, and RF sections, minimizing crosstalk and ground noise. The PCB layout segregates high-current traces and uses wide copper pours to limit voltage drop and heat generation. Each subsystem includes local decoupling capacitors to mitigate transient current spikes during RF transmission or processor wake events. These measures ensure that the device maintains stable voltage regulation and electromagnetic cleanliness even under dynamic load conditions.

Battery selection is based on a 7.4 V lithium-ion pack with a nominal capacity of 3000–4000 mAh, regulated down to subsystem voltage levels. Assuming an average system draw of approximately 6–7 W during continuous use, the device can sustain operation for 45–60 minutes per charge, suitable for prototype testing and short deployment scenarios. In future iterations, implementing power gating, adaptive sensor polling, and radar duty cycling could extend runtime substantially without sacrificing functionality.

Through these architectural and design decisions, the 2HADRAD system remains within its defined 10 W constraint while ensuring safe, efficient, and

thermally stable operation. The deliberate separation of high-power RF components from low-power sensor subsystems, coupled with robust power management and regulation strategies, ensures that the device delivers reliable performance within realistic energy limitations for a field-deployable search-and-rescue prototype.

4.1.8. Thermal Constraints

The 2HADRAD system must maintain stable operation across a wide range of environmental conditions while preventing excessive internal heat buildup from its high-power electronic components. Because it integrates both low-power sensors and high-frequency radio subsystems, attention to thermal design and heat dissipation is essential for reliability, user safety, and sensor accuracy. The device is engineered to operate within a 0 °C to +70 °C ambient range, consistent with typical electronic-grade operating limits, while individual components such as the radar and power regulators are rated for up to +85 °C junction temperature.

The most thermally demanding component in the system is the TI IWR6843AOP radar module, which dissipates approximately 2.0–2.5 W of power during active transmission and onboard signal processing. This heat is concentrated within a compact 9 × 6 cm board area containing both RF front-end and DSP cores. To manage this thermal load, the radar PCB section is mounted to a copper ground plane with thermal vias that transfer heat into internal copper layers and the aluminum mounting plate of the enclosure. This conduction path distributes heat away from the sensitive RF components and reduces localized temperature rise. The enclosure itself serves as a passive heatsink, enabling the device to operate continuously without active cooling while maintaining surface temperatures below safe handling limits.

Additional heat contributors include the ESP32-WROOM microcontroller (~0.5 W) and the DC-DC converters powering the radar and UWB subsystems. These regulators are positioned near ventilation cutouts within the enclosure to promote natural convection. The PCB layout isolates high-dissipation components from temperature-sensitive sensors such as the barometer and magnetometer, which can experience accuracy drift due to thermal gradients. By maintaining a minimum spacing of 20 mm and using local copper shielding, thermal coupling between these subsystems is minimized.

To ensure safe operation under load, the firmware continuously monitors internal temperature sensors located on the radar module and the microcontroller. If the measured temperature exceeds predetermined thresholds (typically +80 °C for the radar and +70 °C for the MCU), the system can automatically enter a reduced-power mode by lowering radar duty cycle or suspending nonessential tasks. This thermal throttling mechanism prevents overheating and extends component lifespan.

Through a combination of passive conduction, strategic component placement, and firmware-based temperature management, the 2HADRAD maintains compliance with its thermal design constraints. This ensures continuous, safe, and reliable operation even during extended use in confined or elevated-temperature environments, critical for field-deployable search-and-rescue applications.

4.1.9. Mechanical / Size Constraints

The 2HADRAD is designed as a portable, handheld search-and-rescue device, which requires strict mechanical and dimensional constraints on the internal layout, enclosure design, and component integration. To remain practical for field use, the total system, including the radar, microcontroller, sensors, and power source, must fit within a compact, lightweight housing that can be comfortably held, mounted, or carried by a single operator. The enclosure dimensions are targeted at approximately 160 × 100 × 60 mm, balancing internal volume for components and airflow with ergonomic handling and manufacturability using 3D printing or CNC machining.

Mechanical constraints are dictated primarily by the TI IWR6843AOP radar module, which requires unobstructed line-of-sight for 60 GHz transmission and reflection detection. This necessitates precise placement of the radar aperture relative to the front face of the enclosure, avoiding any metallic surfaces or thick dielectric materials that could distort the electromagnetic beam. The radar PCB is aligned flush with a non-metallic window, typically acrylic or polycarbonate, chosen for its low dielectric loss at millimeter-wave frequencies. The enclosure front wall thickness is maintained below 1.5 mm in the radar's emission zone to minimize attenuation, while the remaining walls use reinforced ribs for rigidity. The mechanical mounting of internal components follows a layered architecture. The main PCB, housing the ESP32 microcontroller, power regulators, and sensor connectors, serves as the structural backbone. Above it, the UI PCB, containing an LED, 5-way navigation button, the MAX98357A amplifier and the speaker, is mounted orthogonally to align with the front panel openings. Both boards are secured with nylon standoffs and M2.5 screws, ensuring vibration resistance while avoiding conductive shorting paths. Critical connectors, such as the 60-pin Samtec header between the main board and radar module, are positioned to maintain mechanical stability and consistent impedance alignment without strain or torsion.

The design also accounts for environmental mechanical stressors. The enclosure includes integrated gaskets and sealed interfaces to protect against dust and moisture during field operation. Internal cable routing is minimized and bundled using short flexible jumpers to reduce wear and mechanical noise coupling. To mitigate the effects of handling shock or drops, heavy components like the lithium-ion battery pack and radar board are mounted close to the enclosure's center of mass and cushioned with thin neoprene pads that absorb impact energy without restricting heat dissipation.

Mechanical tolerances are maintained within ±0.2 mm for 3D-printed prototype assemblies to ensure proper alignment of mating surfaces and connectors. The final design must also allow for ease of assembly, maintenance, and component replacement, particularly for sensors or modules that may require recalibration or reconfiguration.

Through adherence to these mechanical and spatial constraints, the 2HADRAD achieves a compact, structurally sound design that supports functional alignment of RF, power, and sensing subsystems. The result is a mechanically robust prototype that balances durability, manufacturability, and ergonomic usability while maintaining the precise physical alignment necessary for reliable radar and sensor performance.

4.1.10. Cost Constraints

The 2HADRAD is designed under a strict academic budget, with a total allowable cost of approximately $560, including all sensing, computing, and power components. The largest cost driver is the TI IWR6843AOP radar module, priced around $175, which represents a large portion of the total budget but is essential for achieving real-time vital-sign detection capabilities. Remaining funds are allocated toward the ESP32-WROOM microcontroller, UWB module, sensors, battery, and 3D-printed enclosure. Design choices prioritize integration of multifunctional modules over discrete components to reduce both cost and complexity. The team also minimizes PCB fabrication and assembly expenses by consolidating subsystems onto shared boards. These trade-offs ensure a cost-effective yet fully functional prototype that meets design requirements within the financial limits of a university project.

4.1.11. Environmental Constraints

Since the 2HADRAD is intended for field operation in search-and-rescue environments, it must withstand humidity, dust, and mechanical vibration. The system is designed for reliable operation between 0 °C and +70 °C, consistent with commercial-grade electronics, and uses environmentally resistant materials such as ABS or PETG plastic for the enclosure. All exposed interfaces are sealed with gaskets or silicone to prevent dust ingress, and sensor ports are protected by mesh vents to allow airflow while maintaining particulate resistance. The lithium-ion battery includes overcurrent and overtemperature protection to prevent degradation in hot environments. Moisture-sensitive components, including the barometric and radar modules, are coated with a thin conformal layer to resist condensation. While the device is not waterproof, it is designed for short-term outdoor exposure, ensuring dependable performance in unpredictable field conditions without compromising portability or safety.

4.1.12. Computational Constraints

The 2HADRAD must execute multiple concurrent tasks, such as, radar signal processing, sensor fusion, display management, and wireless communication, on a limited embedded platform. The ESP32-WROOM, with a dual-core 240 MHz processor and 520 KB of SRAM, serves as the central controller but provides only moderate computational resources compared to desktop or FPGA-based systems. To operate within these limits, data processing is optimized through time-sliced task scheduling, lightweight algorithms, and efficient use of DMA channels for SPI and UART communication. Radar preprocessing is largely offloaded to the IWR6843AOP's on-chip DSP, reducing CPU burden. Nonessential tasks such as display updates or BLE communication are throttled during radar operation to preserve real-time responsiveness. These architectural and firmware-level optimizations allow the system to maintain real-time performance within its finite memory and processing bandwidth, demonstrating efficient embedded computation under constrained resources.

4.1.13. Ethical / Safety Constraints

The 2HADRAD design incorporates safety considerations consistent with IEEE and FCC standards to ensure that all transmitted electromagnetic energy remains within non-hazardous exposure limits. Operating in unlicensed frequency bands, the radar and UWB modules emit at power levels well below thresholds defined for human exposure to electromagnetic fields. From an ethical standpoint, the device is intended solely for life detection and rescue applications, avoiding any intrusive or privacy-violating use of radar sensing on individuals. All sensing data are processed locally, with no cloud transmission, preserving confidentiality and aligning with responsible data handling practices. Electrical safety is addressed through reverse-polarity protection, current limiting, and fused battery inputs to prevent overheating or short circuits. By adhering to recognized electrical and ethical standards, the 2HADRAD prioritizes human safety, responsible use of technology, and compliance with societal expectations for ethical engineering conduct.

CHAPTER 5

5.1. Overview

Since its initial release in November 2022, ChatGPT has evolved into one of the most widely adopted AI assistants for academic and professional applications. With the development of GPT-5, the model introduced enhanced reasoning, contextual understanding, and technical depth, making it a powerful companion for engineering research. Going into the early stages of Senior Design I, I anticipated that ChatGPT would become one of my primary tools for researching hardware components, comparing specifications, and assisting in documentation. While it certainly proved helpful, my experience revealed both its strengths and limitations in technical design contexts. When using ChatGPT for research, I

typically began my prompts by clearly outlining my goals and expected outputs, such as tables comparing component specifications, summaries of datasheets, or explanations of circuit behavior.

Across our team, every member approached AI assistance differently. Some of us relied on it for literature review and formatting, while others focused on code optimization, simulation setup, or part selection. To better understand how these models performed under similar conditions, we compared ChatGPT with Claude, developed by Anthropic, and Perplexity AI, an emerging research focused assistant. Claude excelled in long form reasoning and clarity, often producing detailed technical explanations ideal for report writing. Perplexity AI, on the other hand, was notably effective for real-time data retrieval and source verification, making it valuable for up-to-date product comparisons and component availability checks. By analyzing their outputs using identical prompts, our team evaluated each model's accuracy, organization, and relevance. The following sections summarize how each member incorporated these AI tools into their workflow and how the insights gained from them contributed to the efficiency and technical depth of our overall project.

5.2. Advantages and Limitations

5.2.1. Advantages

ChatGPT offers several advantages for our team. It helps quickly summarize academic papers and datasheets on radar, IR, and inertial sensors, providing a solid foundation for component selection and system design. Additionally, ChatGPT streamlines drafting the abstract, introduction, and system descriptions by paraphrasing and formalizing text while maintaining technical accuracy. It also improves the quality of reports by identifying unclear phrasing, redundant sections, and inconsistencies, enhancing both readability and professionalism. During the early design phases, ChatGPT serves as a technical tutor by offering simplified explanations of complex concepts like Doppler radar theory, UWB pulse generation, and signal conditioning, which reduces the time spent cross-referencing multiple textbooks.

5.2.2. Limitations

While ChatGPT proved valuable throughout the research and design process, several limitations were identified during its use. The most prominent issue involved fact verification, as the model occasionally returned inaccurate technical specifications such as component power ratings, MCU clock speeds, or frequency ranges. These discrepancies required consistent cross-checking against official datasheets to ensure design accuracy. Additionally, the model lacks real-time data or simulation capability, meaning that while it can explain expected circuit behavior conceptually, empirical validation still had to be performed manually in LTspice and MATLAB. The quality of responses also depended heavily on prompting style, for example, detailed questions often yielded highly relevant answers, whereas vague or generalized prompts resulted

in off topic or incomplete information. Another noted drawback was the risk of overreliance, as its convenience for writing and summarization could discourage deeper analytical reasoning if not balanced with sound engineering judgment. Finally, the use of AI-generated text introduced copyright and authorship concerns, since large language models draw from broad data sources that may not always be properly attributed. As a result, any material obtained from ChatGPT was treated as an assistive reference rather than an original or citable source, ensuring academic integrity throughout the project.

5.3. Case Studies

5.3.1. Case Study 1: Power Management Optimization

Please refer to Appendix E, [1a], [1b], and [1c] for the Case Study 1 given prompt and responses from ChatGPT and Claude.

This case study was designed to test both models' abilities to generate technical recommendations for optimizing power efficiency in an embedded system using dual-rail Li-ion regulation. ChatGPT provided a concise and practical analysis, emphasizing synchronous buck-boost converters, low quiescent current operation, and firmware strategies such as duty cycling and deep-sleep scheduling. Its response demonstrated strong alignment with the power management goals of the project, offering clear design-level insights that could be directly implemented.

Claude, in contrast, produced a broader systems-level response, prioritizing long-term energy balance, dynamic voltage scaling, and intelligent radar activation based on motion triggers. While both models arrived at similar conclusions regarding converter efficiency and power gating, Claude's answer read more like a white paper, highlighting lifecycle monitoring and hardware-level optimization. ChatGPT's structured, implementation-driven style was more immediately actionable for our design phase, whereas Claude's added context for future optimization potential.

5.3.2. Case Study 2: Signal Filtering in Noisy Environments

Please refer to Appendix E, [2a], [2b], and [2c] for the Case Study 2 prompt and responses from ChatGPT and Claude.

This question evaluated the models' understanding of digital signal processing within the context of a radar-based life-detection system. ChatGPT's response was grounded in classical filter design, suggesting Butterworth and Chebyshev filters for noise suppression, as well as Kalman filtering and FFT-based noise analysis for dynamic signal isolation. Its focus on the respiration frequency band (0.1–0.4 Hz) aligned well with real physiological data, making it a strong technical reference for our filtering implementation.

Claude's response was far more detailed and research-oriented, offering a layered approach that included clutter removal, adaptive background modeling, range gating, and the incorporation of machine learning classifiers to distinguish genuine human signatures from environmental noise. While ChatGPT excelled in explaining practical filter architectures, Claude provided more advanced and multidisciplinary techniques that could be applied in future versions of the system. Together, they reinforced both foundational and modern approaches to radar signal processing.

5.3.3. Case Study 3: Communication Protocol Selection

Please refer to Appendix E, [3a], [3b], and [3c] for the Case Study 3 prompt and responses from ChatGPT and Claude.

This prompt aimed to compare each model's reasoning in selecting an optimal communication interface between the radar baseboard and the ESP32 microcontroller. ChatGPT identified SPI as the most balanced solution for speed, reliability, and ease of wiring, noting its superior data rate and deterministic timing compared to I²C. It also suggested UART as a secondary channel for debugging and low-speed control tasks.

Claude supported this recommendation but elaborated further on potential scalability, suggesting that CAN bus or RS-485 could be explored in later iterations for environments requiring longer cable runs or improved EMI resistance. Its focus on redundancy and flexibility highlighted a more modular communication approach. Both models ultimately aligned on SPI as the best option for the prototype stage, validating the design direction taken by the team.

5.3.4. Case Study 4: Enclosure Material Selection

Please refer to Appendix E, [4a], [4b], and [4c] for the Case Study 4 prompt and responses from ChatGPT and Perplexity.

The fourth case study assessed each model's material science reasoning as it relates to radar transparency and environmental durability. ChatGPT recommended polycarbonate-ABS (PC-ABS) as the optimal enclosure material, emphasizing its low dielectric constant, high impact strength, and ease of machining. It also considered IP-rated sealing and conformal coating for additional protection, aligning closely with real-world mechanical design principles.

Perplexity offered a more data-driven response, comparing several dielectric-friendly plastics and citing radar attenuation levels in decibels. It recommended ABS, PETG, and fiberglass-reinforced nylon as viable materials depending on mechanical and environmental demands. Its explanation of composite-layer shielding and UV stability provided additional depth to the discussion. In this case, both AI models effectively validated the selected PC-ABS material,

reinforcing confidence in its balance between RF transparency, strength, and weight.

5.3.5. Case Study 5: Data Visualization on Embedded Displays

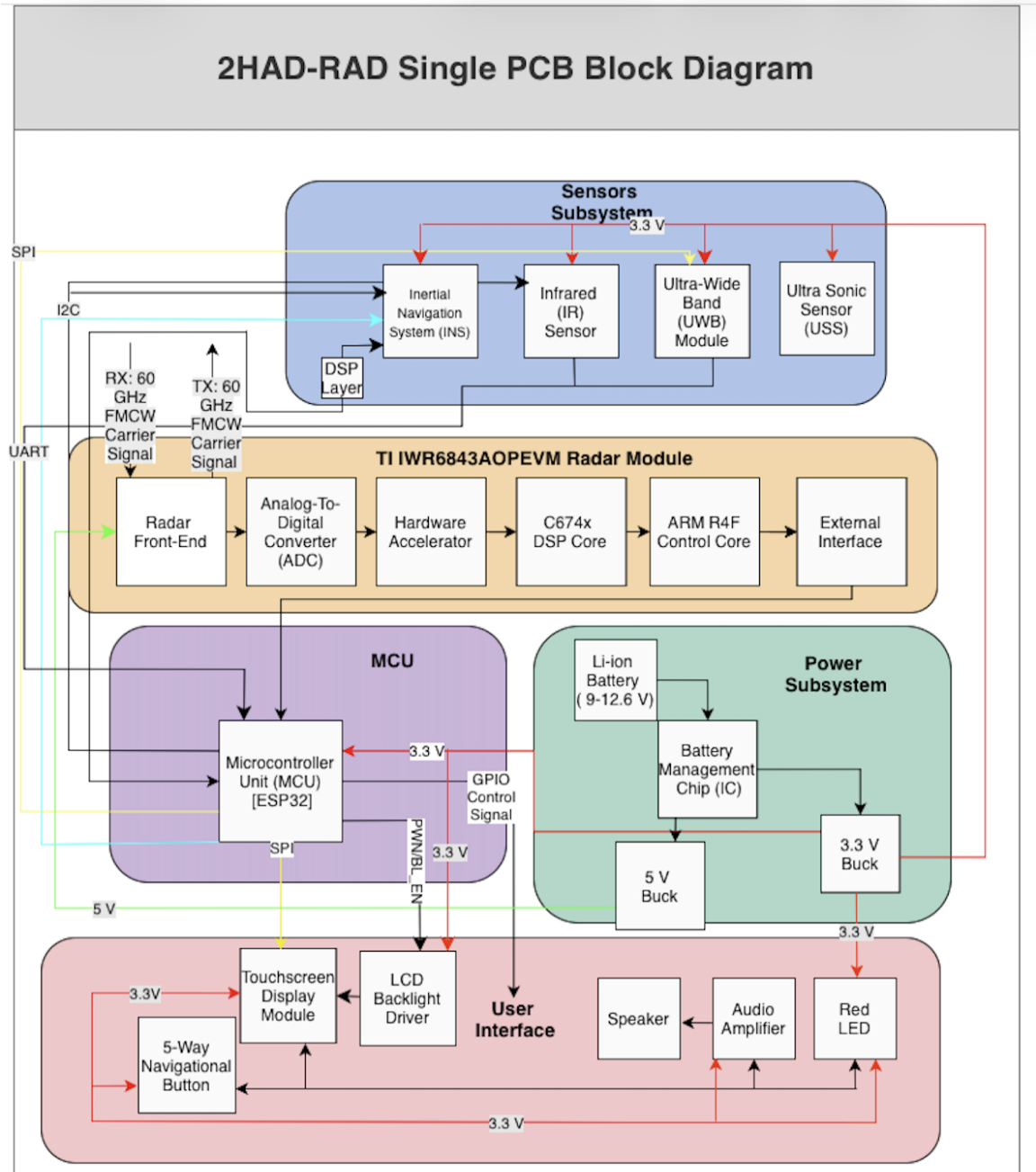
Please refer to Appendix E, [5a], [5b], and [5c] for the Case Study 5 prompt and responses from ChatGPT and Claude.

This case study explored how each model could propose visualization strategies for displaying radar-based motion and respiration data on a compact 4.3-inch TFT screen. ChatGPT's output was practical and implementation-oriented, proposing waveform and bar graph combinations, adaptive brightness control, and color-coded motion indicators. Its suggestions closely matched embedded GUI design best practices and emphasized readability and responsiveness under real-time constraints.

Claude provided a more human-centered response, describing a heat-map style interface with trend graphs, signal confidence indicators, and color-coded alerts to support quick interpretation by field users. It also discussed optimizing refresh rates to match radar update cycles and minimizing redraws to conserve power. Collectively, both responses demonstrated strong design intuition, with ChatGPT emphasizing functionality and Claude prioritizing usability and situational awareness.

CHAPTER 6

6.1. Hardware Block Diagram



Division of Labor		
Subsystem	**Primary Lead**	**Secondary Support**
Power	Ryan Andersen (EE)	Bailey Hitt (EE)
Sensors	Shannon Hankinson (EE)	Bailey Hitt (EE)
Radar	Ryan Andersen (EE)	Shannon Hankinson (EE)
MCU	Shannon Hankinson (EE)	Joshua D'Souza (CpE)
User Interface	Bailey Hitt (EE)	Shannon Hankinson (EE)

Figure 6: Hardware Block Diagram

6.2. Sensor Subsystem

In figures 7–10, the sensor subsystem comprises the primary sensing modules: the MPU9250 IMU and BMP280 barometer (custom INS stack), the Qorvo DW3000 UWB transceiver, and the Adafruit MLX90640 IR array sensor, each interfacing with the ESP32-WROOM-32E over I²C, SPI or UART. Each module directly interfaces with the ESP32-WROOM-32E microcontroller, which acts as the central processing and communication unit of the system. The main sensor subsystem board is mounted on a single PCB with power and signal routing designed to minimize noise, maintain consistent voltage regulation, and ensure synchronization across all data lines.

6.2.1. Radar Module

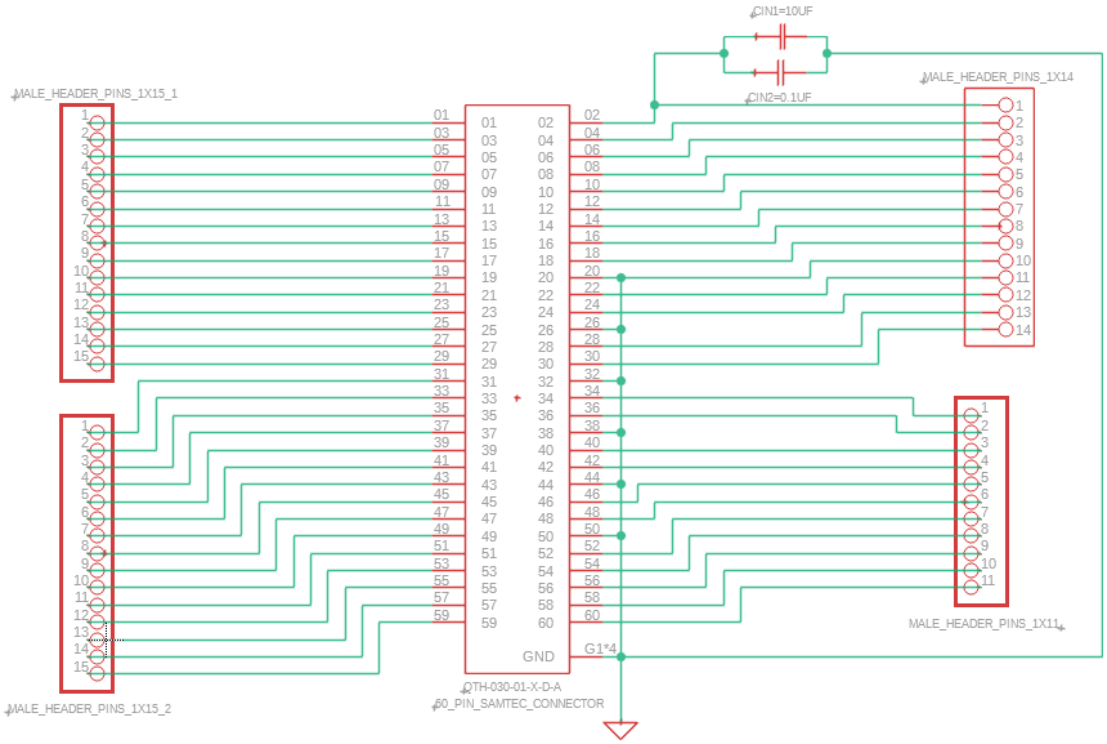


Figure 7: Radar Module PCB Schematic.

6.2.2. INS Module

The original INS design specified the M5Stack M135 module, which integrates a u-blox NEO-M9N GNSS receiver, Bosch BMI270 IMU, BMM150 magnetometer, and BMP280 barometer on a single module communicating over I²C (SDA on GPIO21, SCL on GPIO22) and UART for GNSS streaming. The schematic reflects this original selection. During the build phase the M135 was lost, and the INS subsystem was redesigned using discrete components: an MPU9250 9-axis IMU (accelerometer, gyroscope, and magnetometer) and a standalone BMP280 barometric pressure sensor, both connected over the same I²C bus at the same GPIO pins shown in the schematic. The MPU9250 and BMP280 footprints are

pin-compatible with the I²C lines on the existing PCB, so no board-level rework was required. GNSS streaming is handled via the time-multiplexed UART2 strategy described in the communication protocols section rather than a dedicated UART from the M135.

Series resistors on all data lines and local 10 µF bulk plus 0.1 µF ceramic decoupling capacitors adjacent to the 3.3 V supply pin are retained from the original design and remain appropriate for the MPU9250 and BMP280 operating conditions.

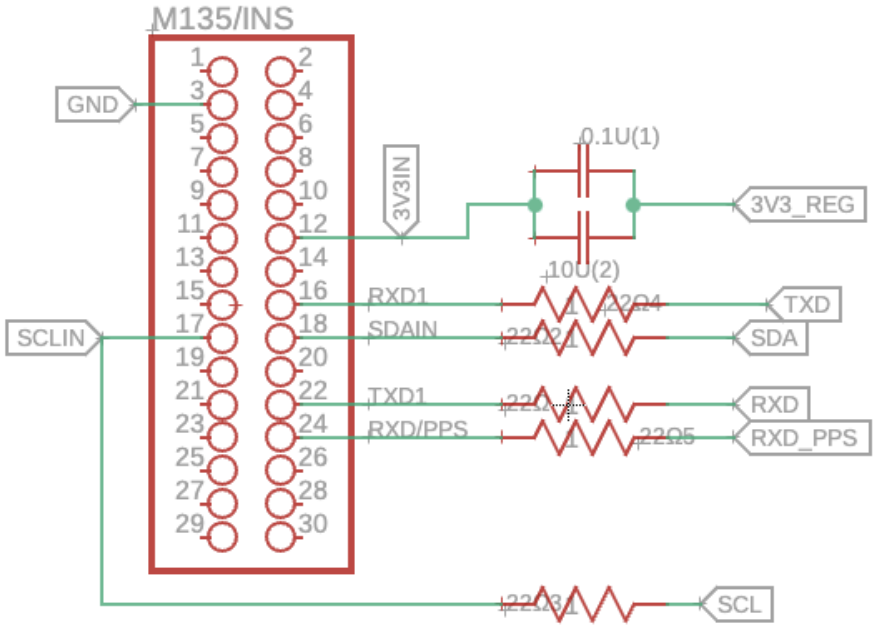


Figure 8: INS module schematic (original M135 design; MPU9250 + BMP280 substituted on I²C lines in final build, GNSS/GPS on UART)

6.2.3. IR Sensor

The Adafruit MLX90640 breakout provides thermal imaging capability via the same I2C bus used by the M135. The sensor's SCL and SDA lines are connected in parallel with the INS, while the pull-up resistors located on the MCU maintain proper bus biasing. Power is provided from the 3.3 V rail with local decoupling. This sensor provides a thermal grid, which will be processed alongside radar and UWB data for motion and human presence detection.

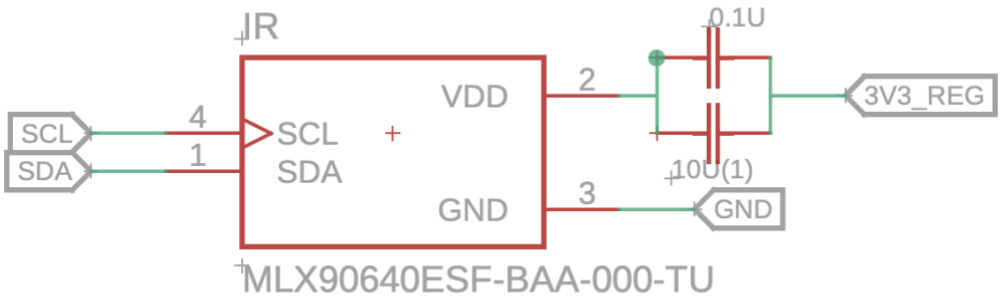


Figure 9: IR sensor schematic.

6.2.4. UWB Sensor

The Qorvo DWM3000 module operates using the SPI interface to communicate with the ESP32. Control signals include IRQ and RST with an optional wake up line. Each SPI line incorporates a small series resistor to reduce reflection along high-speed traces, and local bypass capacitors are used to suppress transient voltage spikes near the UWB module. The DWM3000 enables precision UWB ranging and time-of-flight measurements within an anchor-tag positioning framework. While the handheld unit operates as a mobile tag, a companion Android application allows smartphones to function as cooperative anchors or tags, enabling responders to track team members and, when available, locate victims' devices. UWB ranging provides absolute distance measurements, while the onboard INS supplies relative motion tracking to maintain positional continuity when anchor coverage is limited. Thermal IR sensing supplements localization by confirming human heat signatures and reducing false positives.

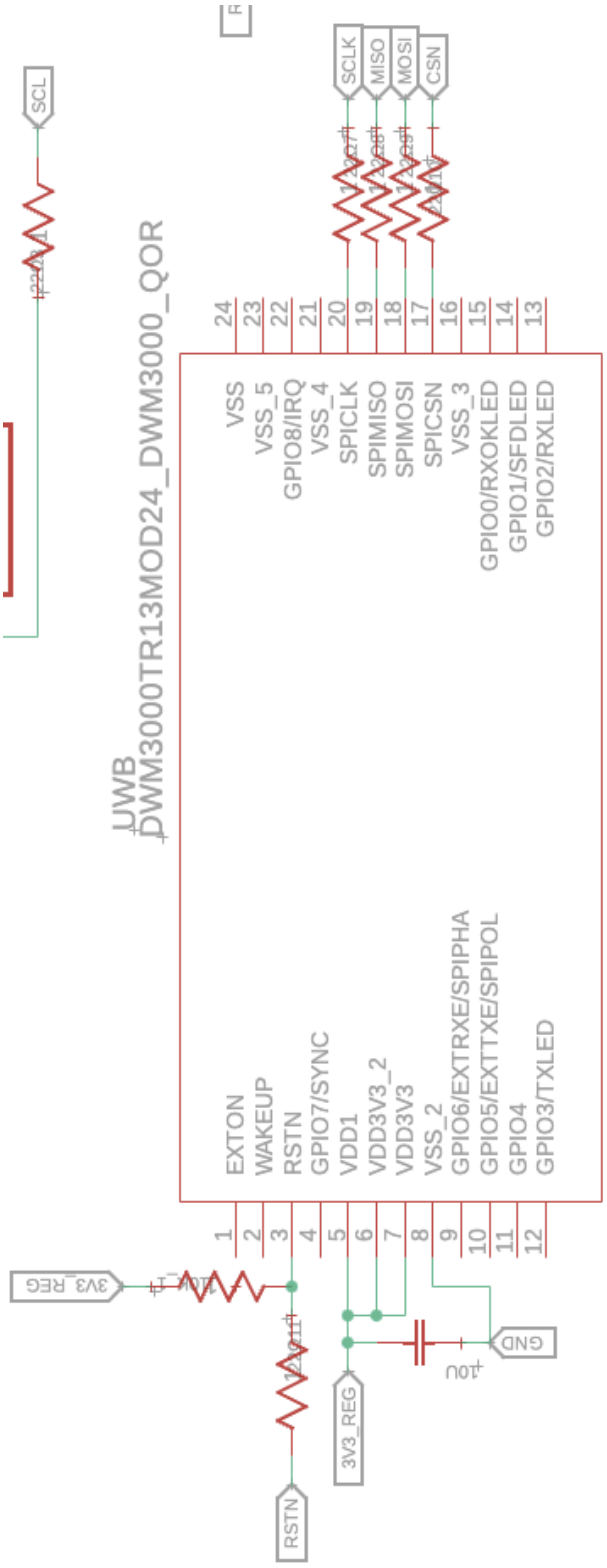


Figure 10: UWB sensor schematic

6.2.5. USS

An ultrasonic sensor provides short-range proximity awareness and supports a radar-style interface that visualizes nearby objects as the user scans the environment with the handheld unit. As the operator moves or reorients the device, the USS continuously measures distance to surrounding obstacles and updates the display to indicate objects within close range. When a nearby target consistent with human presence is detected, the system prompts the operator to confirm using additional sensing modalities, such as mmWave radar or thermal imaging, improving confidence while reducing false positives. The USS also serves a safety function: if an object or person enters a predefined minimum separation distance, radar transmission is suspended to maintain FCC exposure compliance and ensure the required standoff distance is preserved.

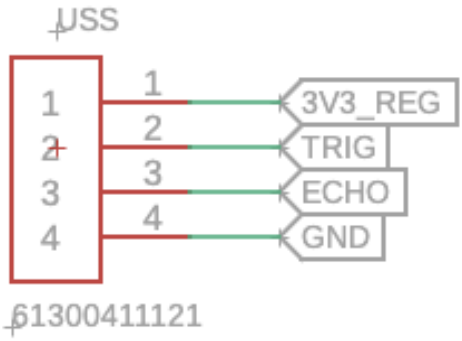


Figure 11: USS schematic

6.3. MCU Configuration

The CP2102 USB-to-UART converter is powered from the 5 V LM2576 buck regulator rail and provides a USB-A interface for programming and debugging during development. The CP2102 handles UART bridging between the host computer and the ESP32-WROOM-32E TX0/RX0 lines (GPIO1/GPIO3), with DTR and RTS control lines connected to the ESP32 EN and IO0 pins to enable automatic bootloader entry from the Arduino IDE without manual button intervention. The ESP32 itself is powered separately from the 3.3 V LM2576 buck regulator rail. The system operates fully on battery power in field deployment with the USB-A connector unpopulated or disconnected. No additional ESD protection or signal conditioning components are present on the

USB lines beyond what is integrated into the CP2102 itself.

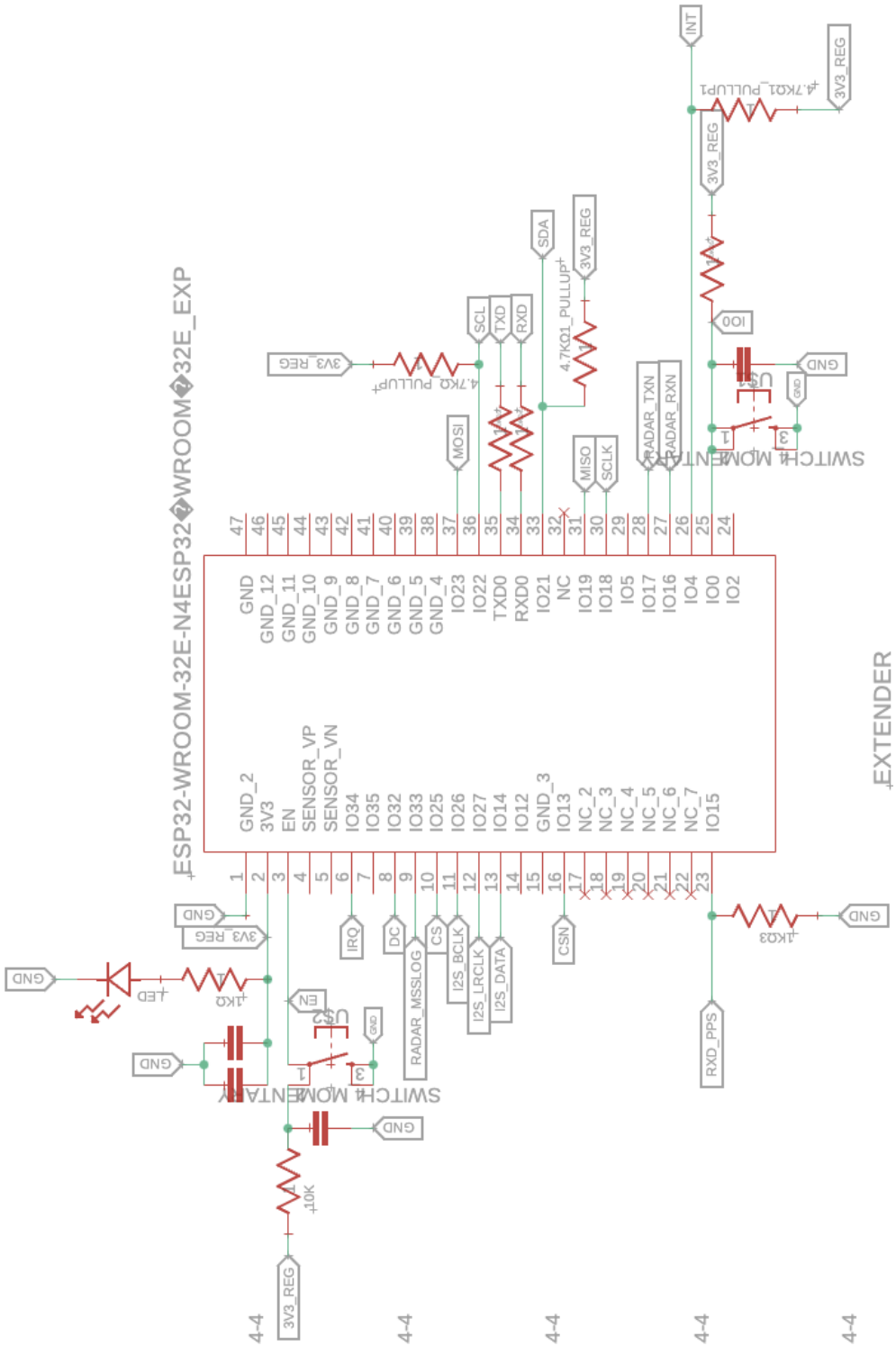


Figure 12: ESP32-WROOM-32E schematic

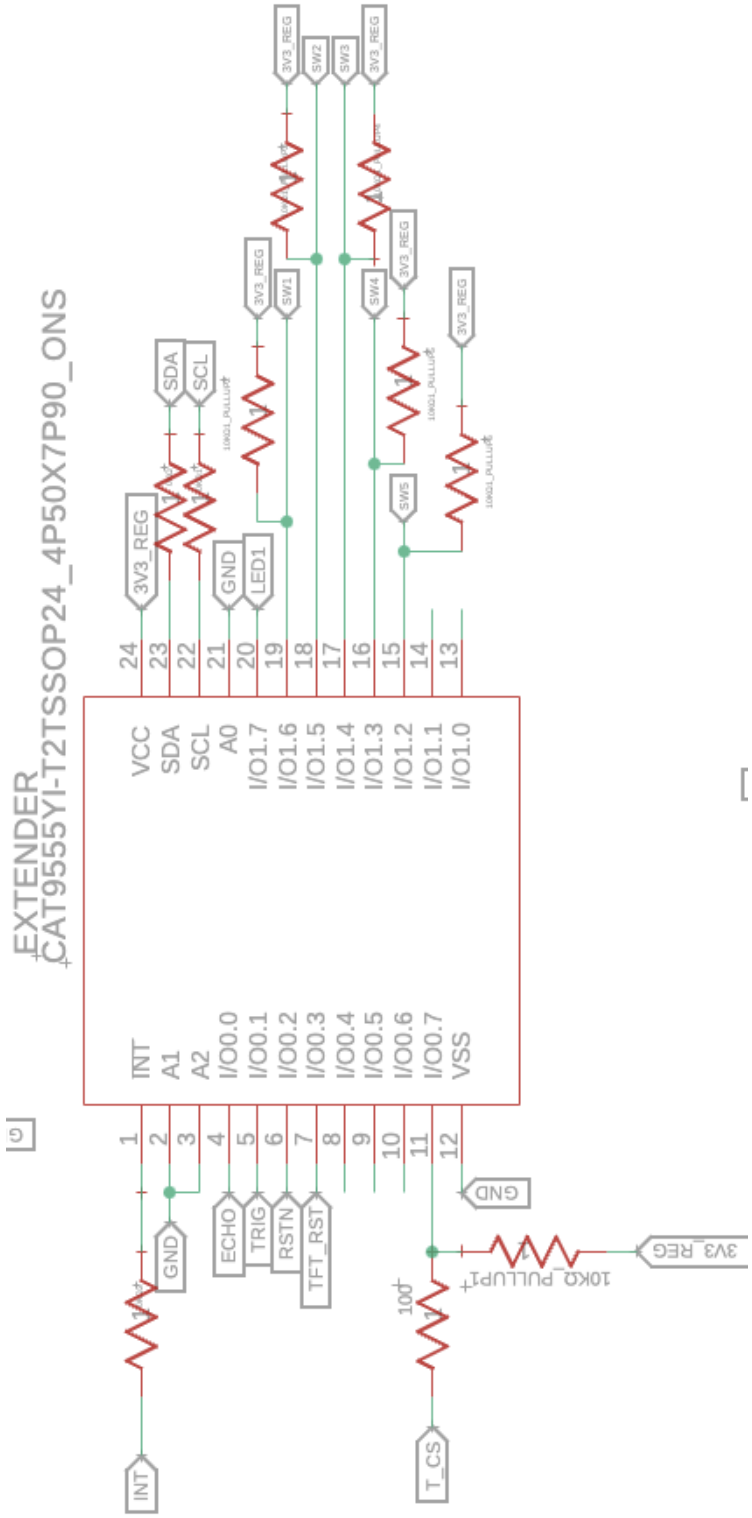


Figure 13: GPIO extender for ESP32-WROOM-32E.

6.3.1. USB to UART

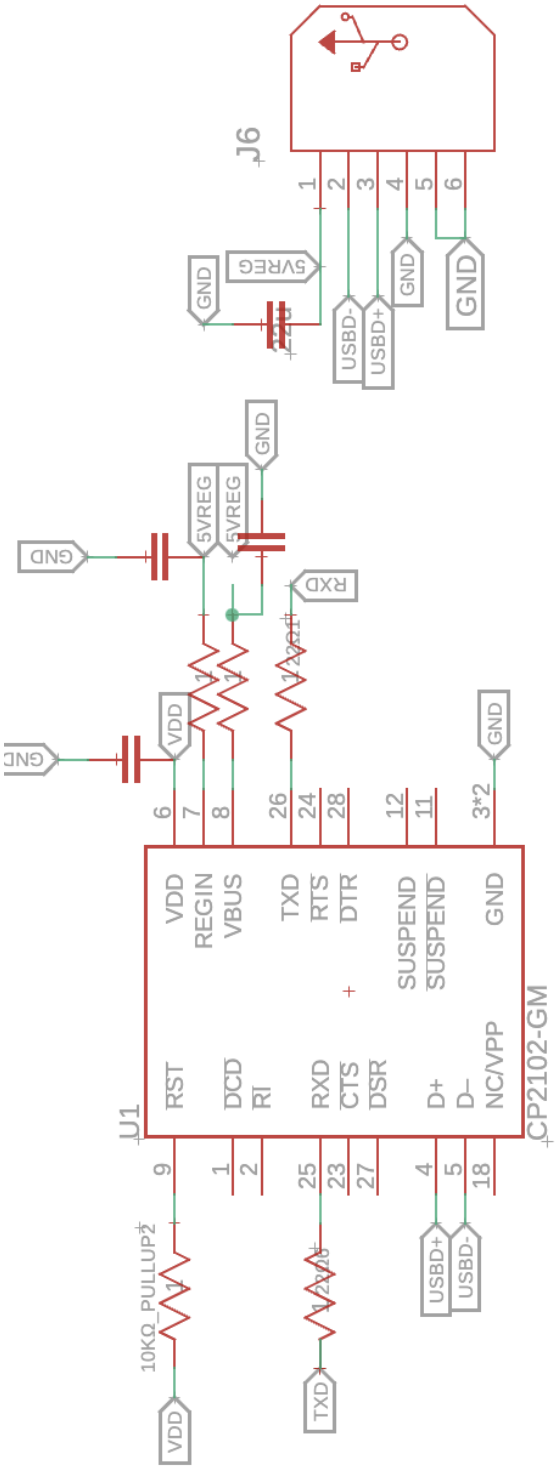


Figure 14: USB to UART schematic.

6.4. User Interface

The user-interface elements shown in Figure 14 integrate three LED status indicators and a single digital pushbutton input directly with the ESP32-WROOM-32E module to provide operator feedback and system control. Each LED is driven from an individual ESP32 GPIO pin using a dedicated series resistor sized for a nominal 2–5 mA indicator current. The LEDs are powered from the regulated 3.3 V rail to ensure GPIO compatibility and prevent back-feeding into higher-voltage domains. Placing each resistor in series with its corresponding LED limits inrush current during switching transitions, reduces stress on the microcontroller pins, and ensures consistent brightness regardless of supply variation. The pushbutton interface uses a momentary, normally open SPST switch that connects the selected GPIO pin to ground when actuated. This pin is configured with the ESP32's internal pull up resistor so the input remains in a stable logic-high state when the button is not pressed. To improve signal integrity and suppress contact bounce, a 100 Ω series resistor and a 0.1 µF shunt capacitor are included at the switch node, forming a compact RC debounce filter placed close to the connector footprint. This filter prevents false triggering and minimizes the effect of mechanical chatter during transitions.

Additional protection features are included to ensure reliable operation of the UI under electrically noisy conditions. Each LED control line incorporates the series resistor as both a current-limiting and transient-damping element, restricting surge currents that may be generated during rapid GPIO toggling. The pushbutton input is protected from ESD and cable-injected noise through its RC filter and through the ESP32's internal ESD structures, which remain within safe limits due to the controlled impedance of the interface. Since the pushbutton is panel-mounted and connected to the main PCB via a two-wire harness, the inclusion of the series resistor at the microcontroller side prevents long cable inductances from coupling fast edges back into the GPIO input path. No external diode protection is required for the LEDs or pushbutton interface, as neither element interfaces with external high-voltage sources, and all signals remain within the 3.3 V logic domain managed by the ESP32. Collectively, these design choices provide a clean, noise-tolerant, and electrically robust user-interface subsystem that maintains predictable behavior across varying operating conditions and ensures long-term reliability in both development and field environments.

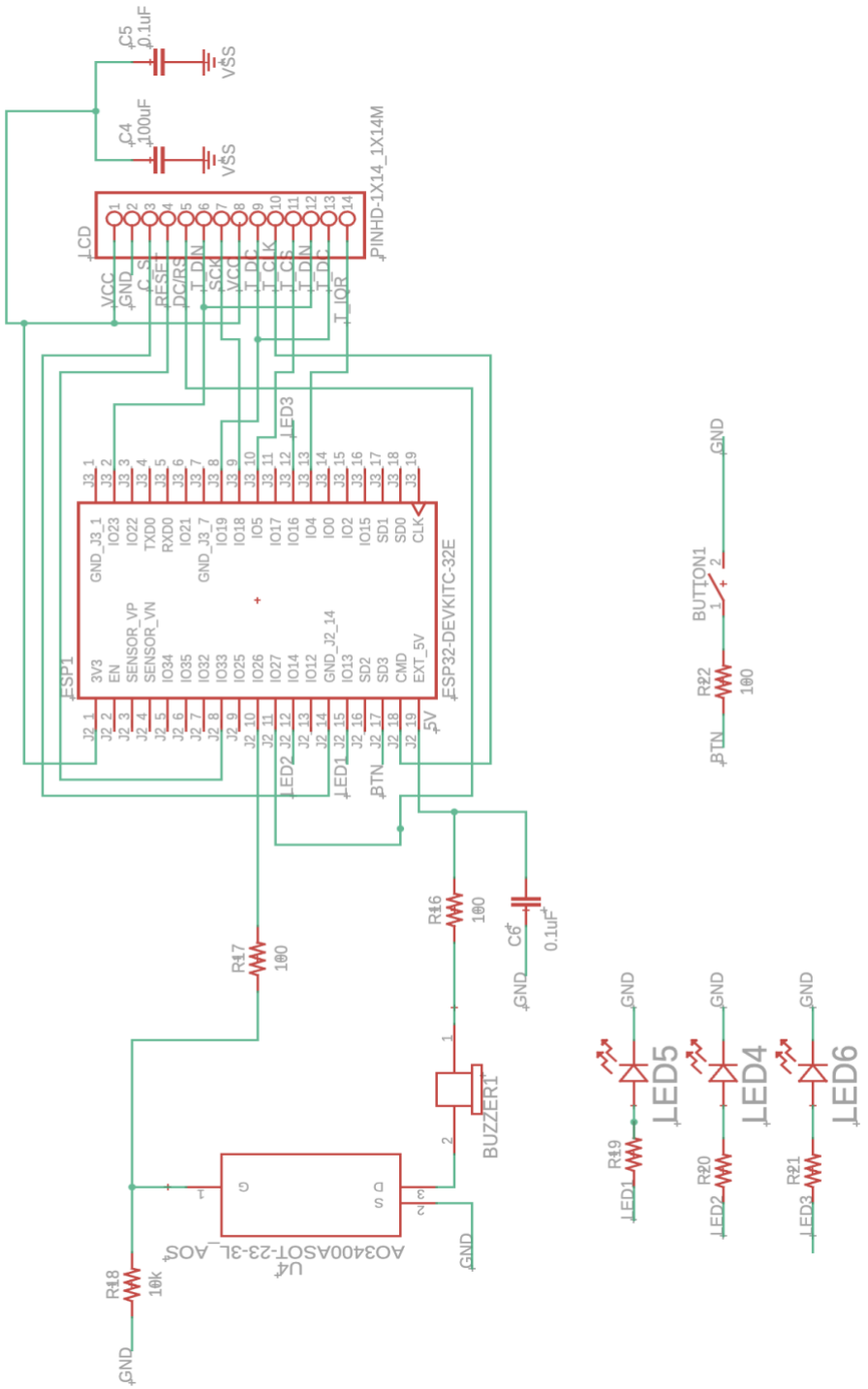


Figure 15: Schematic for UI: Buttons, Indicators, and Display Connections

6.5. Power Subsystem

The schematic shown in Figures 17–19 implements the complete power distribution subsystem for the 2HAD-RAD handheld life-detection device. The system is powered by a multi-cell lithium-ion battery pack, interfaced through a

dedicated battery holder and routed through a fuse-protected input stage on the power carrier board. A resettable fuse provides overcurrent protection at the system input, ensuring safe operation under fault or short-circuit conditions before power is distributed to downstream regulators.

Regulation of the system voltage rails is achieved using two LM2576 switching buck regulators configured for fixed 5 V and 3.3 V outputs. These regulators step down the battery voltage to the levels required by the system's electronics. The 5 V rail supplies higher-power subsystems such as the radar module and other peripherals, while the 3.3 V rail powers the ESP32-WROOM microcontroller, display logic, and low-voltage sensors.

Each LM2576 regulator follows the standard buck converter topology, consisting of an external inductor, catch diode, and output capacitors. The inductors (L1) store and transfer energy during switching cycles, while the Schottky diodes provide a current path during the off-state of the internal switch. Input and output capacitors are placed to reduce voltage ripple and stabilize the converter during load transients. The feedback network sets the output voltage in the adjustable configuration, ensuring consistent regulation under varying load conditions.

Power is distributed across the system using dedicated bus connections for 5 V, 3.3 V, and ground, as shown in the carrier board schematic. These buses provide organized routing to connectors that interface with the radar module, ESP32, and peripheral subsystems. All grounds are tied to a common GND net to maintain a consistent reference and minimize noise and ground potential differences across the system.

Unlike more complex power architectures, this design does not include active power-path management, battery charging circuitry, or firmware-controlled regulation. Instead, it relies on robust, hardware-based regulation and protection to ensure stable operation. This simplified approach reduces system complexity and improves reliability while still meeting the power requirements of the embedded sensing platform.

Overall, the power subsystem provides a straightforward and effective solution for delivering regulated voltages from a battery source, supporting the operation of all major components in the 2HAD-RAD system.

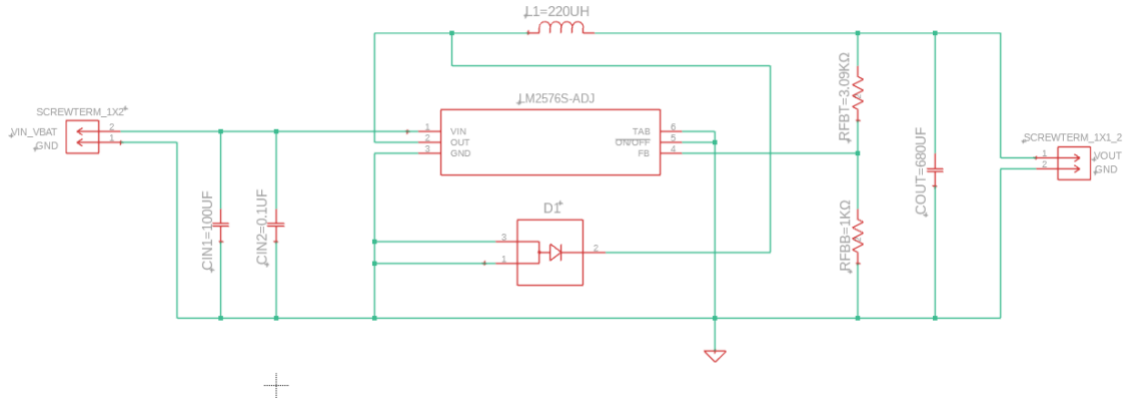


Figure 16: LM2576 5V Regulator

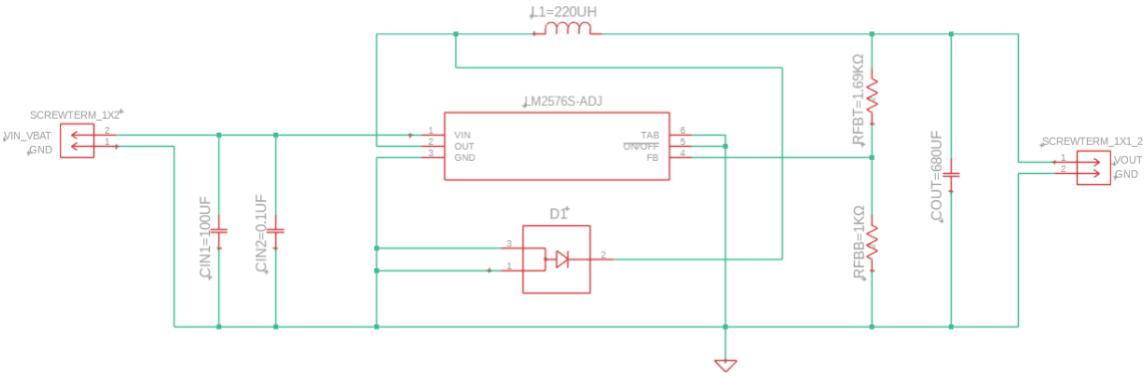


Figure 17: LM2576 3.3V Regulator

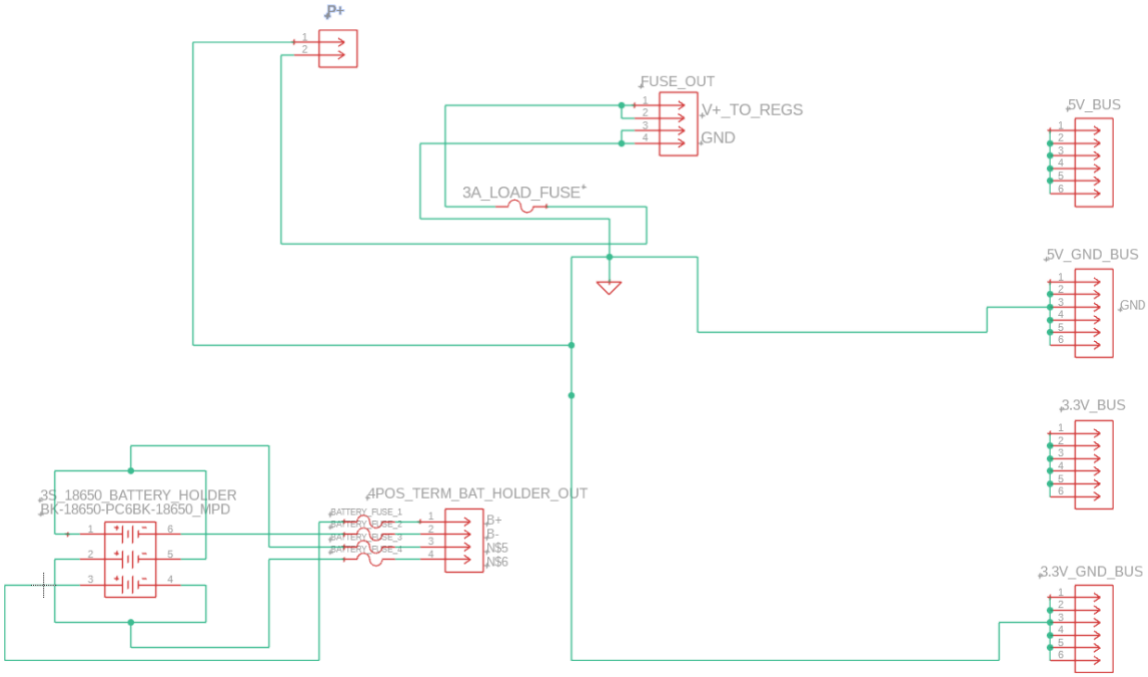


Figure 18: Power System Carrier Board

CHAPTER 7

This project involves the creation of two major software components. One is the base software uploaded on the central ESP32 which handles data from the various sensors and displays it onto the device screen. The second, which is an advanced goal, is the smartphone application that communicates with the MCU through BLE to display data to the user.

7.1. ESP32 Software Design

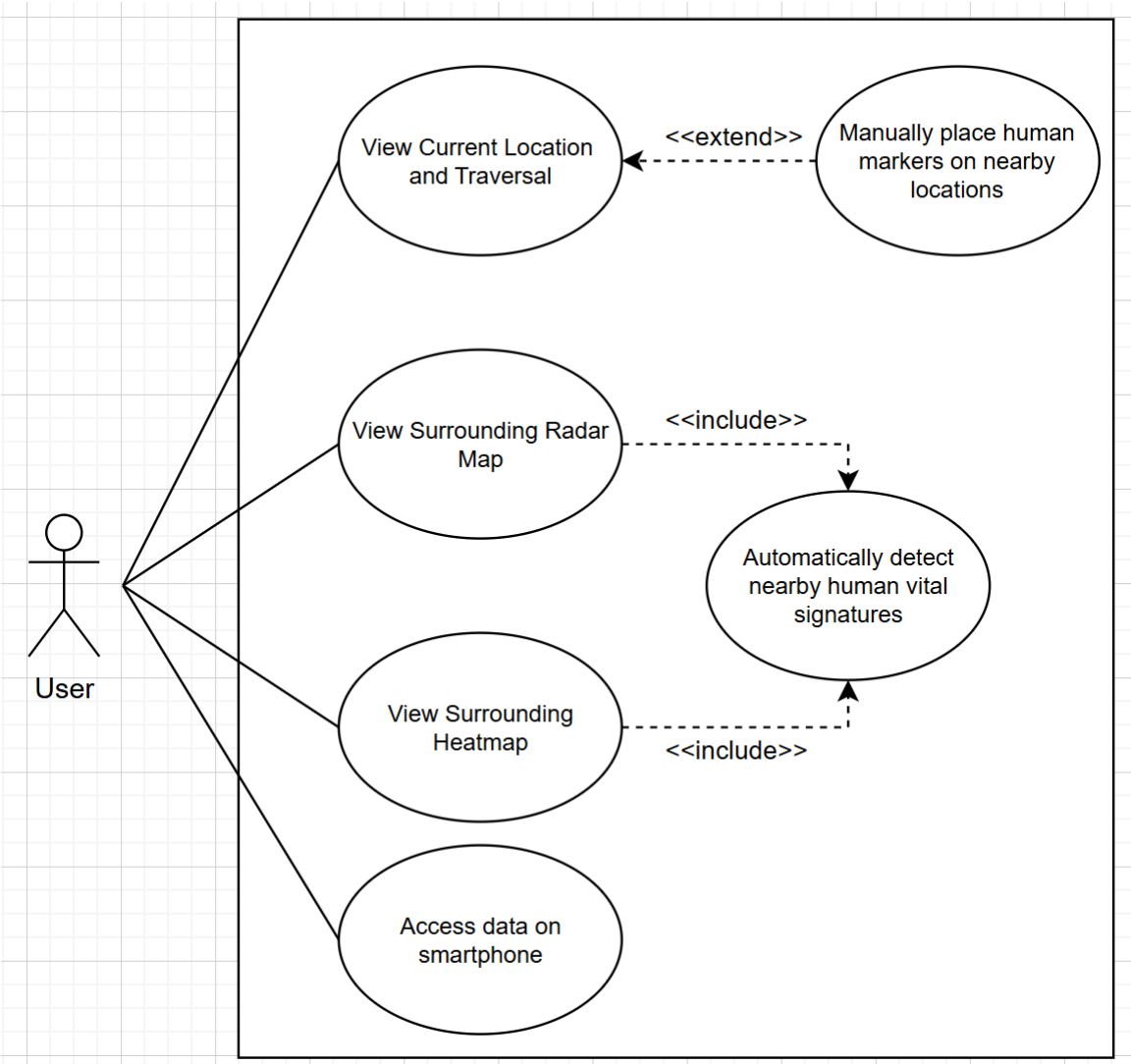


Figure 19: Use Case Diagram

The ESP32 receives signals from the Radar (TI IWR6843AOPEVM), UWB (Qorvo DWM3000EVB), IR (Adafruit MLX90640), and INS (Custom Stack). The ESP32 configures radar parameters so that signals from the radar can be used to detect human frequencies around the device, such as breathing and heartbeat. Signals from the UWB are used to detect human presence in ways the radar may not be able to, such as through walls. The IR sensor sends temperature data to the ESP32 for an additional layer of human detection. The accelerometer, gyroscope, magnetometer, and barometric sensor contained in the INS provide data to the ESP32 for location tracking of the device. An interactive map presents the data from the sensors, displaying the device's position and path. It also includes markers indicating potential human activity and the confidence level of whether a human is present. This is presented through the UI using the 3.2" ILI93414 LCD screen. Audio feedback is provided by a

MAX98357A I²S digital amplifier driving a small speaker, generating distinct alert tones for session events, detections, and proximity alarms.

The device houses pushbuttons corresponding to distinct control functions, such as turning the device on/off, navigating to different operating modes within the main menu, toggling display modes, and acknowledging alerts. Operating modes refer to four different views that the UI provides, discussed in subsequent sections in this chapter. Alerts on the display screen are used to notify the user that a human has been detected, an error has occurred, or when battery gets low. Audio feedback is provided by a MAX98357A I²S digital amplifier driving a small speaker. Distinct audio clips signal navigation confirmation, session start and end, new high-confidence detections, proximity interlock activation, and low-battery warnings. The I²S audio pipeline applies a software filter chain consisting of a single-pole high-pass filter at 120 Hz for DC removal, a three-pole low-pass filter at 3200 Hz for hiss rejection, a noise gate at 0.005 normalized amplitude, and a 0.3 output gain scalar with soft limiting before samples are written to the I²S peripheral. The MAX98357A default gain is 9 dB with the gain-select pin floating and can be increased to 15 dB by connecting a 100 kΩ resistor between the gain-select pin and ground without firmware changes. During audio playback, LCD updates are temporarily paused, and the amplifier is muted when idle to avoid resource contention with the SPI display bus.

7.1.1. User Interface

The UI is organized into four main operating modes: Thermal Map View, INS Map View, Radar Map View, and Scan mode. The user can toggle between these display modes using either of the pushbuttons or the touchscreen. Navigation is handled entirely by the 5-way button read through the CAT9555 I²C GPIO expander - the four cardinal directions move through menu options, and the center button selects. Below the 4 operating mode buttons in the main menu is the button to connect to the accompanying mobile app. The BLE connect option initiates a device scan and connects to the companion iOS or Android application; once linked, all subsequent victim snapshots, fused sensor readings, thermal frames, and radar map data stream in real time without further interaction, with connection status reflected in the persistent top bar. The START button anchors the local coordinate origin and resets the dead-reckoning state, and the MARK button logs the current fused position and sensor state as a confirmed victim snapshot stored in flash for later BLE export.

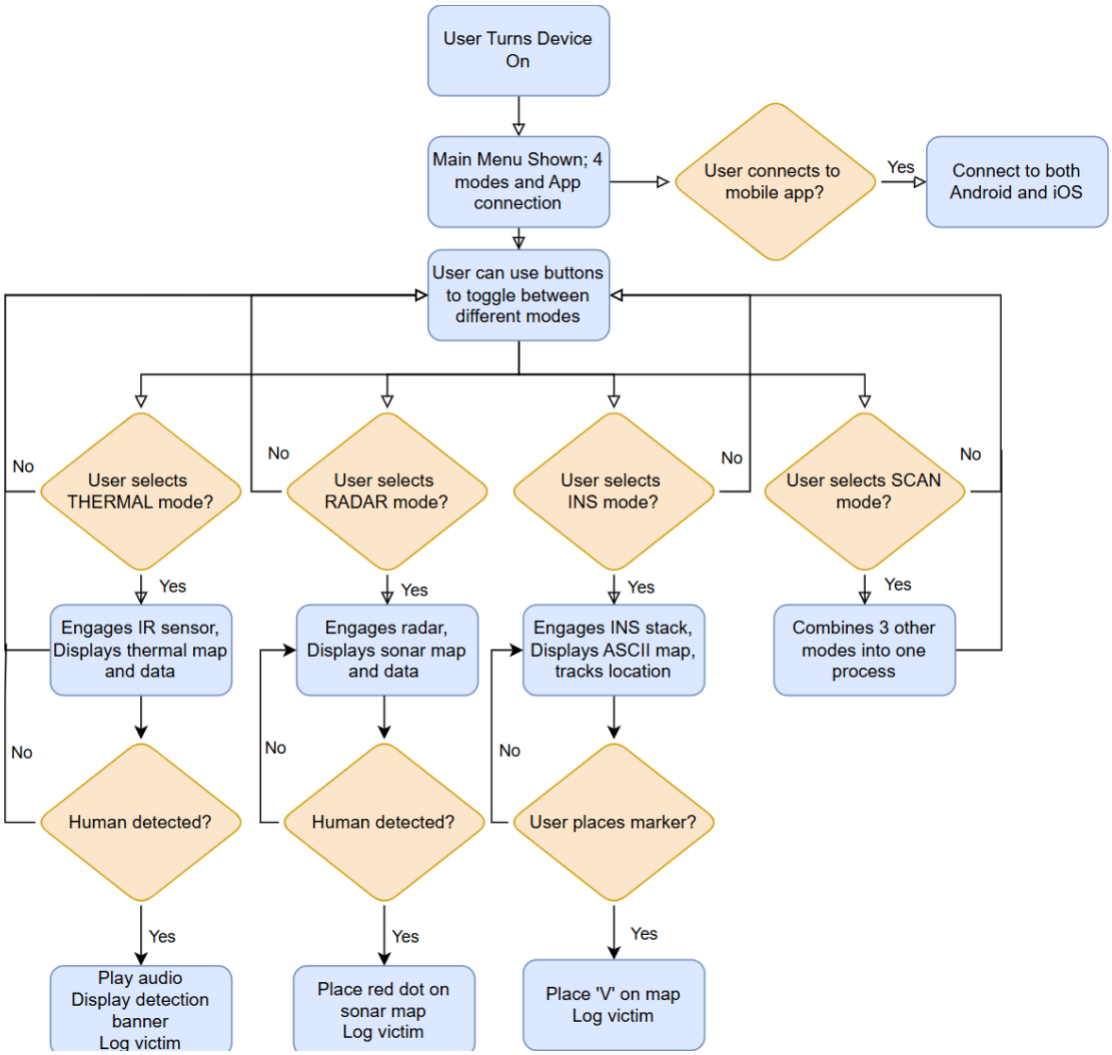


Figure 20: Overall UI Flowchart

7.1.2. Thermal Map Operating Mode

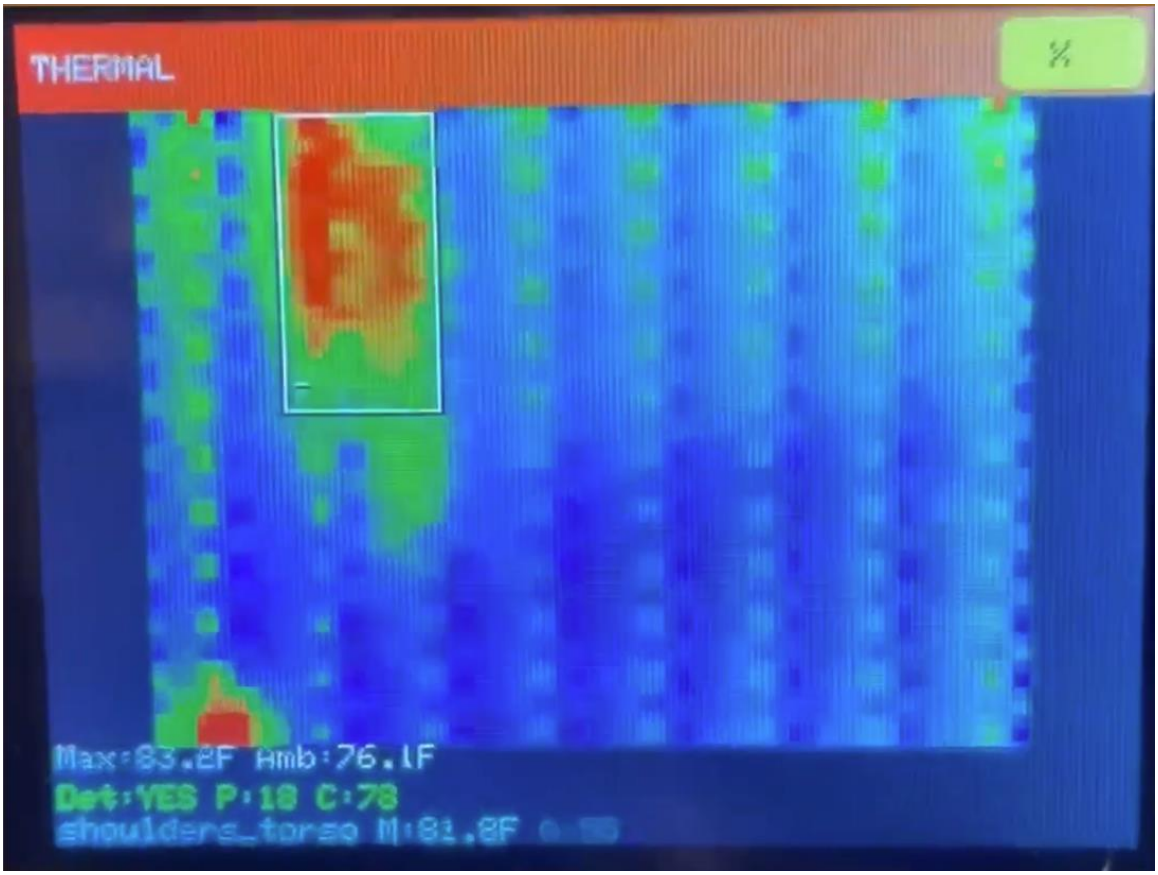


Figure 21: Sketch of Thermal Map

Thermal mode in this program operates as a sophisticated real-time monitoring system that fuses raw sensor data with a multi-stage classification pipeline. When the mode is activated from the main menu, the program first enters a loading state to allow the MLX90640 sensor to stabilize while synchronizing audio feedback. During this setup, the system allocates memory for upscaled image buffers and initializes the ThermalHuman detector state, which includes preparing an adaptive background model that helps distinguish moving human targets from static warm objects in the environment.

The core of the thermal logic is driven by a sensor pipeline that polls the MLX90640 at approximately 10 FPS. Raw temperature data is converted to Celsius and passed through an exponential moving average filter to reduce sensor jitter. This filtered data is then processed by a segmentation algorithm that groups "hot" pixels into blobs based on their contrast against the local background and global scene statistics. These blobs are then analyzed for geometric features--such as aspect ratio, compactness, and vertical energy distribution—to classify them into specific human categories like "HEAD_ONLY," "SITTING," or "FULL_BODY."

For the user interface, the program transforms the low-resolution 32x24 sensor grid into a smoother 64x48 image using bilinear interpolation. It maps these temperatures to a color palette where the bounds are dynamically adjusted based on the current scene's ambient temperature, ensuring that human-level heat signatures are always rendered in high-contrast reds and yellows. To optimize performance on the shared SPI bus, the rendering logic employs a caching system that only updates pixels on the TFT display that have significantly changed color since the previous frame.

Beyond simple visualization, the system maintains a constant alert watchdog. If the classifier identifies a human signature with sufficient confidence over several consecutive frames, the program triggers a "Human Detected" event, firing an audio alert and illuminating an LED. Real-time telemetry, including maximum target temperature and detection confidence, is displayed on an on-screen Heads-Up Display (HUD). Simultaneously, the program can package this data and the raw thermal frames into chunked packets for transmission over BLE to a connected mobile application.

7.1.3. Thermal Classification Pipeline

The thermal classification pipeline is implemented in thermal_human_detector.h and processes each MLX90640 32×24 frame through a multi-stage algorithm designed to overcome the limitations of simple fixed-threshold detection. Fixed-temperature approaches proved unreliable in practice due to wide environmental variability, non-human heat sources such as heaters and sunlight patches, and changing ambient conditions between indoor and outdoor deployments. The full pipeline runs in a dedicated FreeRTOS task to maintain real-time performance without blocking radar, INS, UWB, or display operations.

Preprocessing and upscaling. Each incoming 32×24 frame is first expanded to 64×48 pixels using bilinear interpolation, with each source pixel contributing to its four nearest neighbors in the higher-resolution grid using distance-weighted averaging. A secondary 3×3 weighted spatial smoothing kernel is then applied to suppress sensor noise while preserving thermal edges and maintaining contiguity of detected regions. The upscaled and smoothed frame is passed to the classification stages and simultaneously rendered on the ILI9341 display as the Thermal Heatmap View using a dynamic color palette anchored to the scene ambient temperature.

Adaptive background modeling. Each pixel maintains a dynamically updated background estimate using dual time-constant exponential averaging. A fast update rate (bg_alpha_fast = 0.020 indoors, 0.028 outdoors) applies when the pixel is cool, allowing the background model to track slow environmental drift. A slow update rate (bg_alpha_slow = 0.003 indoors, 0.006 outdoors) applies when the pixel is hot, preventing the model from absorbing a stationary human subject into the background over time. Ambient temperature from the BMP280

barometric sensor is fused into the background model to scale adaptation rates with environmental conditions, reducing false positives from sun-heated surfaces in warm outdoor deployments without requiring manual mode switching by the operator.

Dynamic thresholding. Detection thresholds are computed adaptively from scene statistics rather than fixed temperature values. A pixel is flagged as hot if its temperature exceeds the background estimate by at least bg_delta_c degrees and exceeds the scene 75^{th} percentile temperature plus percentile_delta_c degrees. This dual condition ensures consistent performance across varying ambient temperatures and prevents both missed detections in warm environments and excessive false positives in cool conditions.

Blob segmentation and feature extraction. Hot pixels are grouped into connected regions using 8-connected component labeling. Each blob is characterized by a rich set of features: geometric features including bounding box dimensions, aspect ratio, fill ratio, and edge compactness; thermal distribution features including energy concentration in the top, middle, and lower thirds of the blob, vertical temperature gradient, width taper ratio, and local contrast relative to the background estimate; and temporal features including centroid stability and frame-to-frame persistence. Blobs outside the valid size range (min_blob_area to max_blob_area = 340 pixels) or with fill ratios outside the 0.14–0.90 window are rejected before classification.

Shape-based classification. Valid blobs are scored against seven human body-part silhouette classes: HEAD_ONLY, HEAD_SHOULDERS, SHOULDERS_TORSO, LOWER_BODY, FULL_BODY, SITTING, and LAYING_DOWN. Classification uses a weighted combination of aspect ratio ranges, energy distribution rules, and fill geometry. Standing and upright poses require aspect ratio below standing_aspect_max = 1.35, while LAYING_DOWN requires a width-to-height ratio above laying_aspect_min = 1.75. Temperature is used as a supporting rather than primary criterion: human targets at close range typically exhibit mean thermal values in the 28–36°C range, while a hard upper limit of max_hotspot_temp_c = 45°C rejects heater-like hotspots regardless of shape score.

Temporal smoothing and decision logic. The per-frame classification result is filtered through an exponential moving average with asymmetric time constants: rise_alpha = 0.12 for fast response to new detections and fall_alpha = 0.04 for slow decay to prevent flickering when a subject briefly moves out of the thermal field of view. Hysteresis thresholds of present_on = 0.66 and present_off = 0.50 gate the final thermalHumanDetected flag, requiring the smoothed confidence to rise above 0.66 to assert a detection and fall below 0.50 to clear it. Additional logic handles cold-ambient shape-only fallback, where a strong geometric match is accepted even when thermal contrast is low, and outdoor mode adjustments that tighten thresholds to compensate for higher background clutter. The pipeline

outputs a human presence confidence score, the best matching class label, and the blob centroid position, which are overlaid on the Thermal Heatmap View and transmitted to the companion iOS and Android applications over BLE as part of each victim snapshot.

7.1.4. Radar Operating Mode

Figure 22: Sketch of Radar Mode

The Radar Operating Mode functions as a multi-sensor spatial awareness system that combines mmWave radar data with ultrasonic ranging to provide real-time target tracking and safety monitoring. When initialized from the menu, the system powers on the IWR6843 radar module and begins a dedicated two-phase boot sequence. During Phase 1, the radar runs a 30-second chirp profile optimized for vital sign acquisition, collecting TLV type 1040 breathing and heart rate frames and type 1021 presence flags while the system warms up and stabilizes its background model. At the end of Phase 1, the firmware hardware-resets the module, waits for the boot prompt, then transmits the Phase 2 configuration over the CLI channel and resumes TLV collection on the data channel - transitioning into a chirp profile optimized for target position and distance resolution for the remaining 30 seconds. This two-phase strategy allows the system to acquire physiological evidence before committing to a spatial track.

The mmWave radar logic is handled by a dual-UART pipeline. The program uses a CLI channel at 115,200 baud to send configuration commands and a high-speed data channel at 921,600 baud to receive a stream of TLV packets. A radarPoll() function parses this stream, extracting vital signs (heart rate and breathing rate from TLV type 1040) and target list coordinates from TLV type 1010. To ensure accuracy, the system filters these readings using an EMA (Exponential Moving Average) smoother on both heart rate and breathing rate output, reducing frame-to-frame jitter without introducing the lag of a boxcar average. A median filter on the raw range-bin data further suppresses spike artifacts before the distance estimate is committed. The firmware also explicitly screens for a known 50.2 BPM subharmonic artifact produced by the IWR6843's FFT pipeline, treating heart rate as a presence-corroborating signal rather than a standalone measurement when this artifact is detected. Breathing rate values are accepted within a 5–40 BPM window, and the detection gate requires both a presence flag and valid vitals to assert a human detection, preventing false positives from background reflections alone.

The ultrasonic sensor acts as a critical safety watchdog for the radar. The program polls the USS at approximately 16 Hz, using a median-of-three filter on raw pulse measurements to eliminate jitter before applying a step-limited EMA for final smoothing. If the USS detects any object within a 20 cm threshold, the proxTooCloseLatched flag is set. This immediately triggers a hardware-level shutdown of the radar module via the NRST pin to prevent signal saturation or interference and plays a "Too Close" audio alert. The radar remains held in reset until the path is cleared and the user manually restarts the scan.

The visual sonar map is built as a polar coordinate display centered on the screen. Unlike mechanical radars that use a rotating servo, this program fuses the radar data with the handheld's IMU. It uses the integrated yaw heading (yawDeg) as the angular source for the radar sweep. As the user rotates the device, the program calculates the polar-to-pixel coordinates for a 60-pixel radius representing 3 meters, effectively using the device's physical orientation to sweep the digital environment.

To achieve a classic radar look, the rendering logic implements a phosphor-decay effect. It maintains a shift register of the last six headings (trailYaw). Each frame, the program erases the oldest sweep line and repaints the entire history using a color gradient from bright green to dim green, creating a visual tail that follows the device's movement. After drawing the sweep, the system restores the static grid of three concentric rings and eight radial spokes, ensuring the map UI remains legible over the moving phosphor trail.

When a human signature is confirmed, a red detection blip is rendered on the scope. The blip's distance is derived from the radar's most precise peak range-bin data and is updated dynamically as the user or the target moves. This visualization is supported by a HUD at the bottom of the screen displaying USS

distance, fused heart and breathing rates blended from the local IWR6843 and external BLE vitals from a Radar2BLEClient anchor, the radar-derived target distance in both feet/inches and meters, and a state indicator toggling between SCAN and TRACK to reflect whether a human signature is currently confirmed.

7.1.5. INS Operating Mode

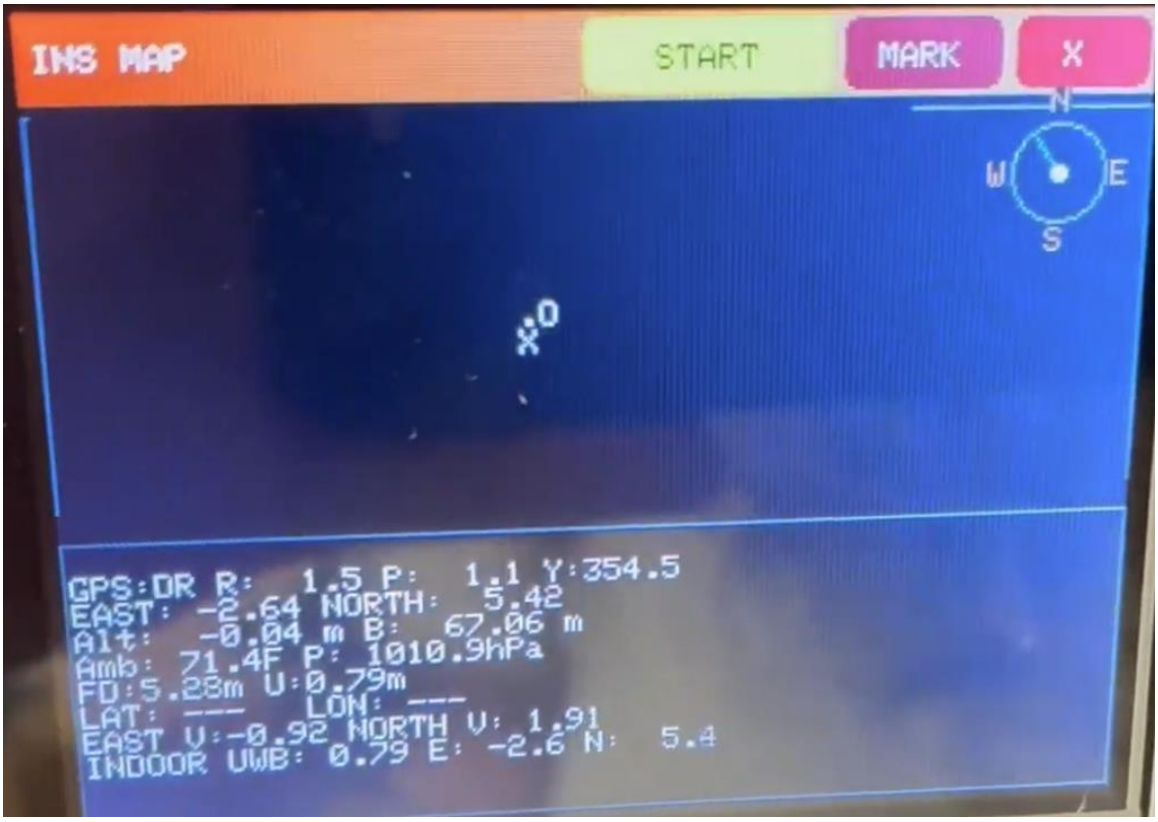


Figure 23: Sketch of INS Operating Mode

The INS (Inertial Navigation System) mode operates as a sophisticated dead-reckoning and spatial tracking engine built around two complementary orientation and position estimation algorithms running sequentially within the same FreeRTOS task at approximately 100 Hz. At the system's core, an MPU9250 IMU is processed through a Madgwick gradient-descent AHRS filter, fusing accelerometer and gyroscope data to produce a stable quaternion orientation that is subsequently converted to Euler angles for display and world-frame rotation. While pitch and roll are derived this way, yaw is maintained separately by integrating the raw gyro-Z axis output minus a calibrated bias to avoid the Madgwick yaw drift often found in magnetically disturbed indoor environments. The system further implements tilt-compensation for the magnetometer, rotating raw magnetic vectors into the horizontal plane based on current pitch and roll estimates. This allows the device to maintain an accurate compass heading at varying angles, serving as a fallback reference and as the sweep angle source for the Radar Map View when valid.

The motion tracking pipeline translates physical movement into coordinate space by rotating the body-frame accelerometer vector into a world-frame (East/North) coordinate system using a 3×3 rotation matrix derived from the current roll, pitch, and yaw estimates. After subtracting pre-calibrated biases and gravity, an EMA low-pass filter is applied to the world-frame acceleration components prior to integration, reducing high-frequency noise before velocity and position are computed via double integration. To combat the inherent drift of inertial sensors, a still-detection watchdog gate monitors the magnitude and variance of total acceleration and gyroscope signals on every frame. If the device remains stationary within a narrow band for a configurable number of consecutive frames, the system forcefully zeros the velocity accumulators to prevent the virtual position from sliding away while the user is standing still.

For absolute distance verification and heightened accuracy, the system utilizes an 8-state Extended Kalman Filter (EKF), defined in handheld_ekf.h, which fuses dead-reckoned data with Ultra-Wideband ranging via a DW3000 module and BMP280 barometric altitude corrections. The state vector tracks X, Y, Z, VX, VY, VZ, YAW, and BARO_BIAS. This EKF runs at approximately 100 Hz driven by IMU samples, applying UWB scalar updates independently per axis at 3.5 sigma innovation gating to reject multipath outliers, while joint Z-axis and bias updates from the barometer use 3.0 sigma gating. The fusion algorithm assigns weights based on confidence: UWB is favored when the signal is fresh and stable, but its measurement confidence degrades quadratically as readings age beyond 1.5 seconds. Specifically, the fusion alpha weighting is set to 0.70 for fresh UWB, 0.50 for aging readings, and 0.25 during INS-only periods. Furthermore, outlier rejection discounts UWB contributions by 70% if the UWB and INS estimates disagree by more than 3 meters within the first 30 seconds of operation.

System initialization begins when the user presses START, executing calibrateIMUQuick to collect 300 samples for accelerometer and gyroscope bias estimation, along with calibrateAltitudeBaseline to collect 12 barometric samples for zeroing the altitude reference. At this moment, the local coordinate origin is set, velocity and position are reset, and the INS Map View track buffer is cleared for a fresh session. The user interface features a graphical compass and a dynamic ASCII-based navigation map rendering a span centered on the user's current position. This map displays breadcrumbs for the travel path, an 'O' for the starting origin, and an 'X' for the current location. Because the TFT display, touch controller, and UWB sensor share a single SPI bus, the program utilizes a semaphore-based mutex (spiBusMutex) and high-speed caching to update only the specific lines of text or map segments that have changed, ensuring the tracking loop remains responsive. Finally, the system acts as a data logger for search-and-rescue operations via a MARK function. This captures Victim Snapshots into a ring buffer, packaging East/North offsets, GPS coordinates (if available), barometric altitude, and ambient temperature with real-time telemetry like heart rate and breathing rate from the radar. This data is transmitted as

chunked UTF-8 packets over BLE to a connected mobile application for remote monitoring.

7.1.6. Scan Operating Mode

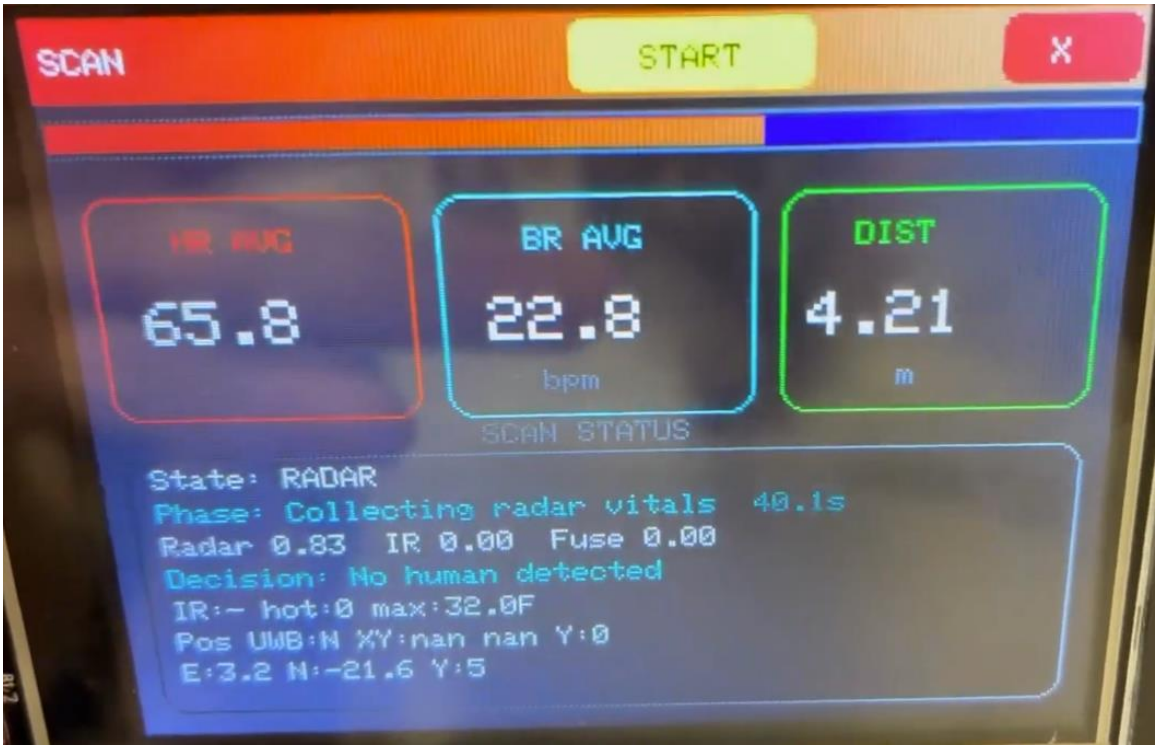


Figure 24: Sketch of Scan Operating Mode.

The scan mode engine is implemented in scan_mode.h and orchestrates the complete multi-sensor detection sequence through a state machine that progresses through five distinct phases: SCAN_INIT, SCAN_RADAR, SCAN_IR, SCAN_POSITION, and SCAN_FUSION. The engine is invoked when the operator presses START and produces a single unified human-presence confidence score from weighted evidence accumulated across all active sensor modalities.

SCAN_INIT: On entry the engine resets all per-scan accumulators, clears the confidence state from any previous session, confirms that the radar, thermal, INS, and UWB subsystems are active and returning valid data, and asserts the USS safety interlock check. If the ultrasonic proximity sensor detects an obstacle within 22 cm the engine holds in SCAN_INIT and does not progress until the path is clear, preventing radar emission in violation of FCC RF exposure requirements.

SCAN_RADAR. The engine collects radar evidence over a configurable window, accumulating TLV type 1021 presence flags, type 1040 vital-sign readings, and type 1010 target position data from the IWR6843AOP. Breathing rate values within the 5–40 BPM acceptance window and heart rate values are

averaged across the window to produce stable per-scan BR and HR estimates. The presence flag and the availability of valid vitals are combined into a radar confidence contribution. Because heart rate estimation on the IWR6843AOP exhibits subharmonic locking behavior clustering near 50.2 BPM regardless of true cardiac rate, HR is treated as a presence-corroborating signal rather than a standalone measurement - the radar confidence contribution requires both a presence flag and valid BR rather than requiring accurate HR. When the anchor BLE link is active, BR, HR, and distance from the Waveshare HMMD anchor radar are averaged with the primary radar readings during this phase, providing a second independent physiological measurement stream.

SCAN_IR: The engine invokes the thermal classification pipeline and waits for a stable thermalHumanDetected result. The pipeline's temporal smoothing ensures that a single noisy frame does not prematurely assert or clear the thermal evidence. The thermal confidence contribution is taken from the smoothed EMA value at the end of the IR phase window, providing a continuous rather than binary input to the fusion step. If the thermal subsystem is paused due to audio playback the engine holds in SCAN_IR until rendering resumes.

SCAN_POSITION: The engine queries the INS and UWB subsystems for the current fused position estimate. The 8-state Extended Kalman Filter is interrogated for the current X, Y, Z position, the associated position covariance, and the UWB confidence weighting in effect at that moment. A high UWB confidence (fresh ranging fix, alpha = 0.70) contributes a stronger position evidence score than an aging or INS-only estimate (alpha = 0.25). The position evidence is used to determine whether the detection is spatially consistent with a prior detection in the same session, which increases fusion confidence when multiple passes converge on the same location.

SCAN_FUSION: The evidence scores from radar, thermal, and position phases are combined using a weighted sum. Radar and thermal evidence carry the highest weights as independent physical confirmation modalities; position evidence acts as a consistency multiplier rather than a primary detection signal, boosting confidence when spatial context supports the sensor readings and discounting it when the device is stationary or position uncertainty is high. When the fused confidence score exceeds the detection threshold, humanDetected is asserted, the red LED on GPIO4 is illuminated, the appropriate audio clip is triggered via the MAX98357A I²S amplifier, and the current fused position and full sensor state are logged as a victim snapshot in flash. The snapshot includes the radar BR and HR averages from SCAN_RADAR, the thermal class label and confidence from SCAN_IR, the UWB and INS position from SCAN_POSITION, and a timestamp. The engine then transitions to SCAN_DONE and returns to idle, ready for the next START press.

Real-time rendering callbacks are invoked throughout all phases so that the active display mode - Raw Data View, INS Map View, Radar Map View, or

Thermal Heatmap View - continues updating without waiting for scan completion. This ensures the operator retains full situational awareness during the scan sequence and is not presented with a frozen display while the engine accumulates evidence.

7.1.7. Smartphone App Software Design

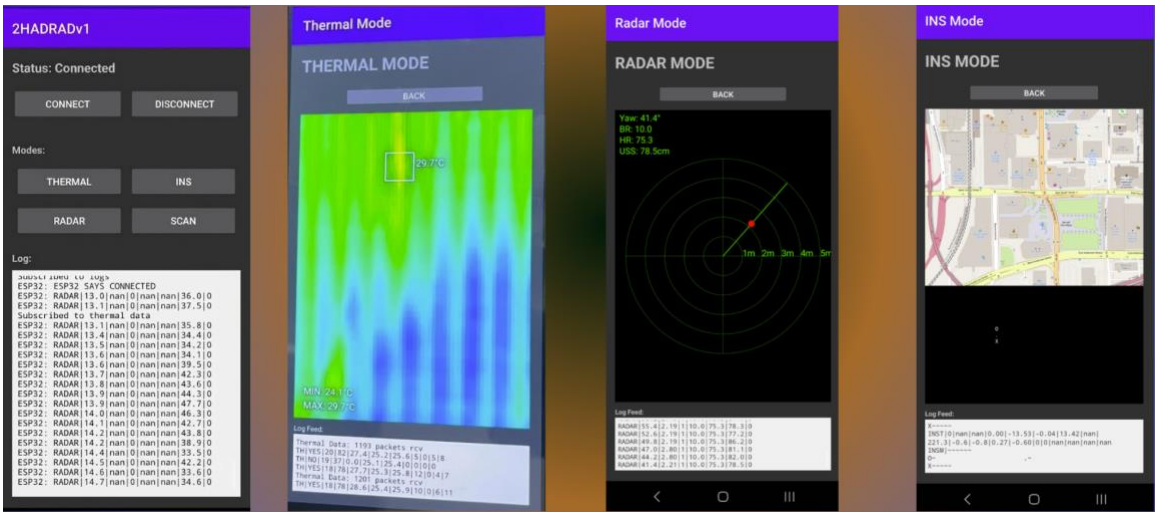


Figure 25: Sketches of Smartphone App UI.

The Android app is a sophisticated sensor-fusion and visualization platform designed to interface with external hardware via Bluetooth Low Energy (BLE). The app serves as a mobile command center, processing high-frequency raw data from multiple sensors--including thermal, radar, and inertial units--to provide real-time environmental awareness. Built using modern Android components like Jetpack Compose and custom views, the architecture is designed to handle the high throughput required for fluid sensor rendering and complex data reassembly.

The initial setup revolves around the BleManager, which handles the discovery and pairing of the hardware module. Upon launching the app, users must grant necessary permissions for Bluetooth and Location (required for BLE scanning). The BleManager establishes a connection and manages the MTU (Maximum Transmission Unit) constraints by implementing a robust packet reassembly logic. Since large data frames (like a 776-byte thermal frame) exceed standard BLE packet sizes, the app utilizes a chunking protocol where each packet is indexed, validated, and reassembled in a buffer before being passed to the respective visualization mode.

Reflecting the 4 operating modes of our main device, the app has those modes too. The Thermal Mode utilizes the ThermalMapView to render data from a sensor like the MLX90640 (32x24 resolution). This mode is highly optimized for visual clarity; it applies an Exponential Moving Average (EMA) for temporal smoothing to reduce flicker and a 5-tap spatial filter to soften noise. To provide a

high-resolution feel, the raw 32x24 grid is upscaled to a 64x48 grid using bilinear interpolation. The UI includes dynamic "hotspot" tracking, where a bounding box automatically anchors to the highest temperature detected, displaying real-time Celsius values alongside a color-coded heatmap.

The Radar Mode (driven by the RadarMapView) provides spatial awareness by visualizing distance and velocity data for objects in the vicinity. This mode typically interprets radar signatures to plot "blips" on a polar or Cartesian grid, allowing the user to see the relative position of moving or stationary targets. By offloading the heavy rendering to a custom View or Compose canvas, the app maintains a high frame rate, ensuring that the movement of detected objects remains fluid and responsive to the actual environment.

The INS Mode, implemented in the InsActivity, focuses on movement and orientation. This mode processes data from an Inertial Measurement Unit (IMU), tracking pitch, roll, and yaw, as well as linear acceleration. This data is critical for understanding the orientation of the sensor hardware itself. The UI likely features a digital horizon or a 3D representation of the device, allowing users to monitor the physical state and movement patterns of the hardware in three-dimensional space. This mode also integrates Geospatial Mapping, leveraging the osmdroid library to provide an offline-capable map interface. This mode combines GPS data from the mobile device with the sensor data processed in the other modes. By overlaying radar or INS data onto a real-world map, the application allows for "breadcrumb" tracking and geographic tagging of detected objects. This fusion of local sensor data with global positioning creates a comprehensive tactical display, suitable for navigation and environmental monitoring in diverse terrains.

7.1.8. iOS Companion Application

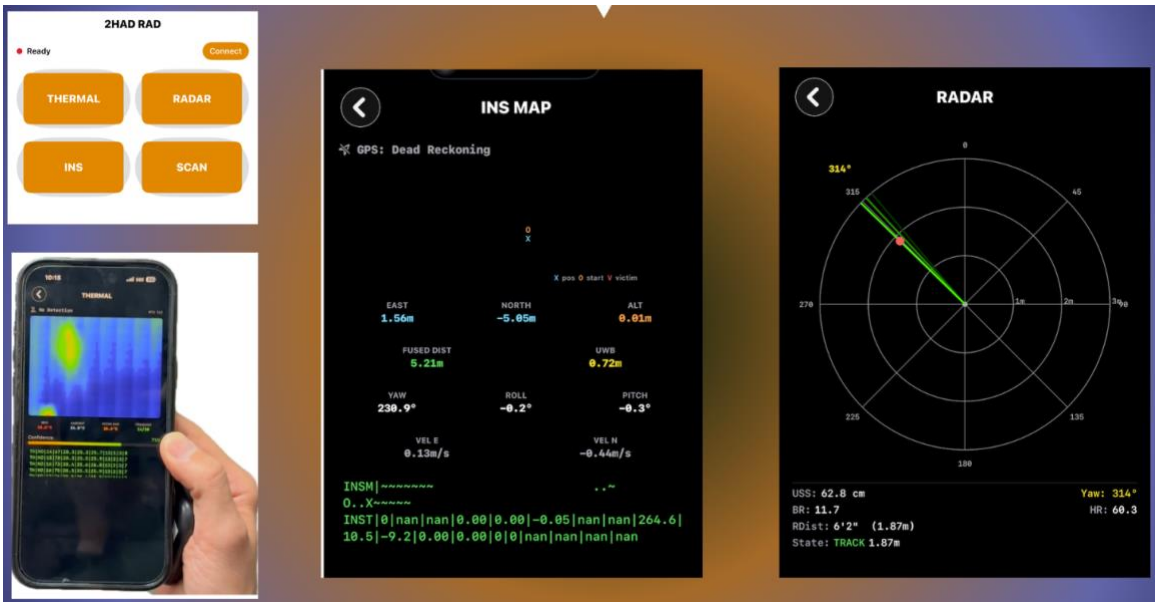


Figure 26: iOS App Interface.

A native iOS companion application was developed in Swift using the SwiftUI framework as a direct functional equivalent to the existing Android application. Development began by decompiling the Android APK to extract the BLE service and characteristic UUIDs, GATT callback flow, and data dispatch logic, which were then re-implemented using Apple's CoreBluetooth framework.

The central BLEManager class conforms to ObservableObject and manages the full CoreBluetooth lifecycle: scanning for the ESP32 peripheral by device name, connecting, requesting MTU negotiation, discovering services, and subscribing to characteristic notifications. On iOS, Client Characteristic Configuration Descriptor (CCCD) writes for notification subscription are handled automatically by CoreBluetooth via setNotifyValue, unlike Android where the descriptor write must be performed manually. This difference required no change to the ESP32 peripheral firmware - both platforms subscribe to the same characteristic UUIDs and receive identical notification payloads.

Incoming characteristic data is dispatched by UUID. String payloads carrying radar vitals, INS state, and scan mode results are appended to a shared log observable that all views observe for updates. Raw byte payloads from the thermal characteristic are parsed as a 32×24-pixel array and rendered as a live thermal heatmap using a SwiftUI Canvas, applying the same dynamic color palette used on the ILI9341 display - cool values blue, mid-range cyan and green, warm values yellow and orange, and the hottest pixels red.

The application provides five dedicated views: a radar and proximity view displaying live mmWave vital-sign estimates and USS distance state, a thermal heatmap view rendering the live MLX90640 frame, an INS navigation view showing dead-reckoning position and heading, a scan mode results view displaying the fused confidence score and detection outcome from the most recent scan cycle, and a connection management view for initiating and monitoring the BLE link to the ESP32. All five views update in real time from the shared BLEManager observable without requiring manual refresh.

The application was verified to connect to the ESP32, receive real-time sensor data across all five views, and render the 32×24 thermal frame correctly. Both iOS and Android applications are confirmed operational against the same ESP32 firmware and BLE characteristic set, ensuring that rescue teams using either platform receive identical victim data, thermal frames, radar map information, and fused sensor readings from the primary device.

7.1.9. Anchor Software Unit

The anchor unit runs a custom firmware in Radar2.ino on the Arduino Uno R4 WiFi, implementing an independent vital-sign extraction pipeline that operates concurrently on three peripherals: the DWM3000EVB UWB module on SPI, the Waveshare HMMD 24 GHz mmWave radar on UART (Serial1, D0/D1), and a

16×2 I²C LCD for local status display. All three buses operate independently with no arbitration required between them.

Frame parsing and distance tracking. Incoming HMMD frames are parsed byte-by-byte through a four-state machine (WAIT_HEADER → READ_LEN → READ_DATA → READ_TAIL). Each 35-byte payload yields a presence result byte, a raw distance in centimeters, and 16 gate energy values at 70 cm spacing. Distance is refined by a nearest-strong-gate selector that weights gate energy to resolve the closest high-energy gate, then
smoothed by an alpha-beta tracker with outlier rejection. A subject distance lock engages once breathing is confirmed, with adaptive tolerance tightening as the classifier history grows to prevent background targets from contaminating the vital-sign buffers.

Breathing rate extraction. Gate energy values are log-compressed using log1p and pushed into a 128-sample ring buffer at 10 Hz. The buffer feeds a three-band respiratory classifier covering Adult (0.15–0.38 Hz, 9–23 BPM), Child (0.28–0.62 Hz, 17–38 BPM), and Infant (0.50–0.95 Hz, 28–60 BPM) ranges. A Hann-windowed FFT with parabolic peak interpolation, subharmonic preference, and double-harmonic suppression extracts the dominant respiratory frequency in each band. A fast path operating on 64 samples runs every 500 ms to achieve earlier lock before the full 128-sample buffer fills. Band selection uses a hysteresis-gated classifier requiring five consecutive votes to switch
bands, preventing oscillation between adjacent classifications. A Kalman filter with SNR-adaptive measurement noise smooths the final BPM output, and a distance-aware plausibility gate rejects infant-band detections at ranges where infant signal return is physically implausible.

Cardiac rate extraction. Heart rate is extracted from a separate 256-sample ring buffer at the same 10 Hz sample rate, providing a 25.6-second analysis window. The pipeline applies breathing harmonic notching in the frequency domain with adaptive notch width that widens at lower SNR and for higher harmonics, then performs an inverse FFT to recover the notch-filtered time-domain signal. Autocorrelation is computed over the physiologically valid lag range for the active age band (Adult: 65–105 BPM, Child: 70–130 BPM, Infant: 100–160 BPM), with a narrow search window biased toward the expected lag from the prior valid reading. Alias detection rejects 2× and 3:2 ratio aliased lags by comparing autocorrelation strength at the candidate and expected lags. A bootstrap accumulator requires three consecutive consistent readings before the Kalman filter is seeded, preventing early false locks. The FFT peak and autocorrelation result must agree within 12 BPM; disagreement triggers fallback to whichever estimate is closer to the prior reading.

Motion gating and signal quality. A variance gate rejects frames where signal variance exceeds a distance-adaptive threshold, preventing motion artifacts from contaminating the vital-sign buffers. An energy spike detector monitors for log-

energy jumps exceeding 1.5 per sample and clears the breathing buffer after three consecutive spikes. The subject distance lock releases after four seconds of sustained divergence from the locked range, triggering a full reset of both breathing and cardiac state. All three operations - breathing analysis, fast breathing, and cardiac analysis - run sequentially in the anchor main loop with non-blocking budget-limited UART draining to maintain responsiveness across all three peripherals.

BLE transmission and fusion. Breathing rate, heart rate, and UWB-measured distance are transmitted from the anchor to the primary ESP32 over BLE at each completed analysis cycle. On the primary device, the fusion layer averages the anchor BR, HR, and distance with the corresponding IWR6843AOP readings when the anchor BLE link is active, providing a second independent physiological measurement stream from the fixed anchor vantage point. The 16×2 LCD on the anchor displays live breathing rate, heart rate, and UWB distance locally so anchor placement and signal quality can be verified during deployment without requiring the primary device.

CHAPTER 8

8.1. PCB

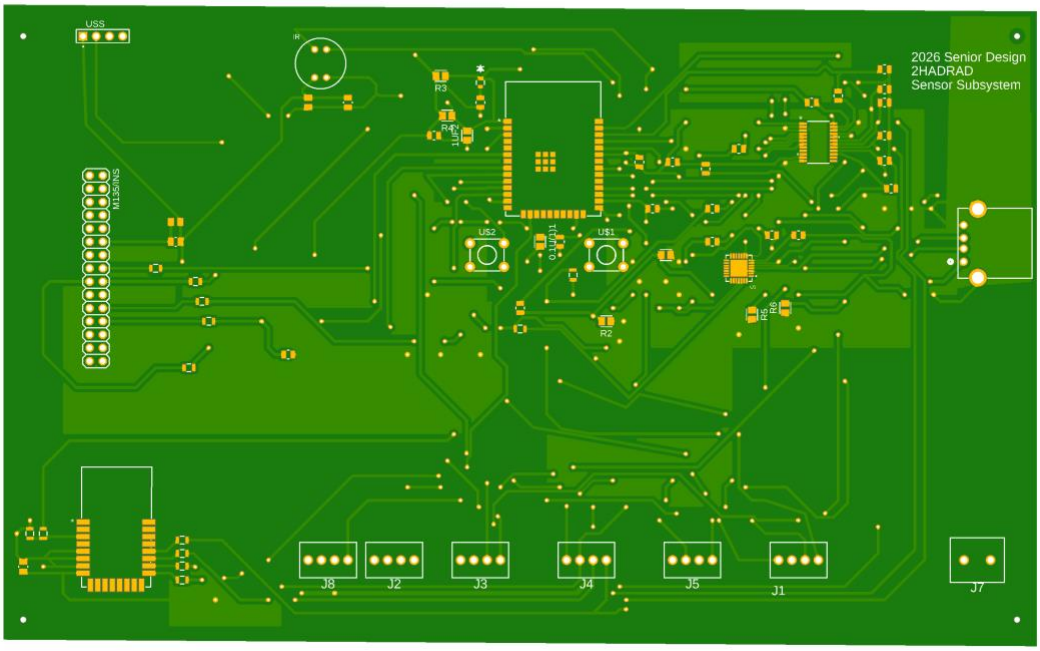


Figure 27: Sensor subsystem PCB for USS, UWB, IR, INS and MCU

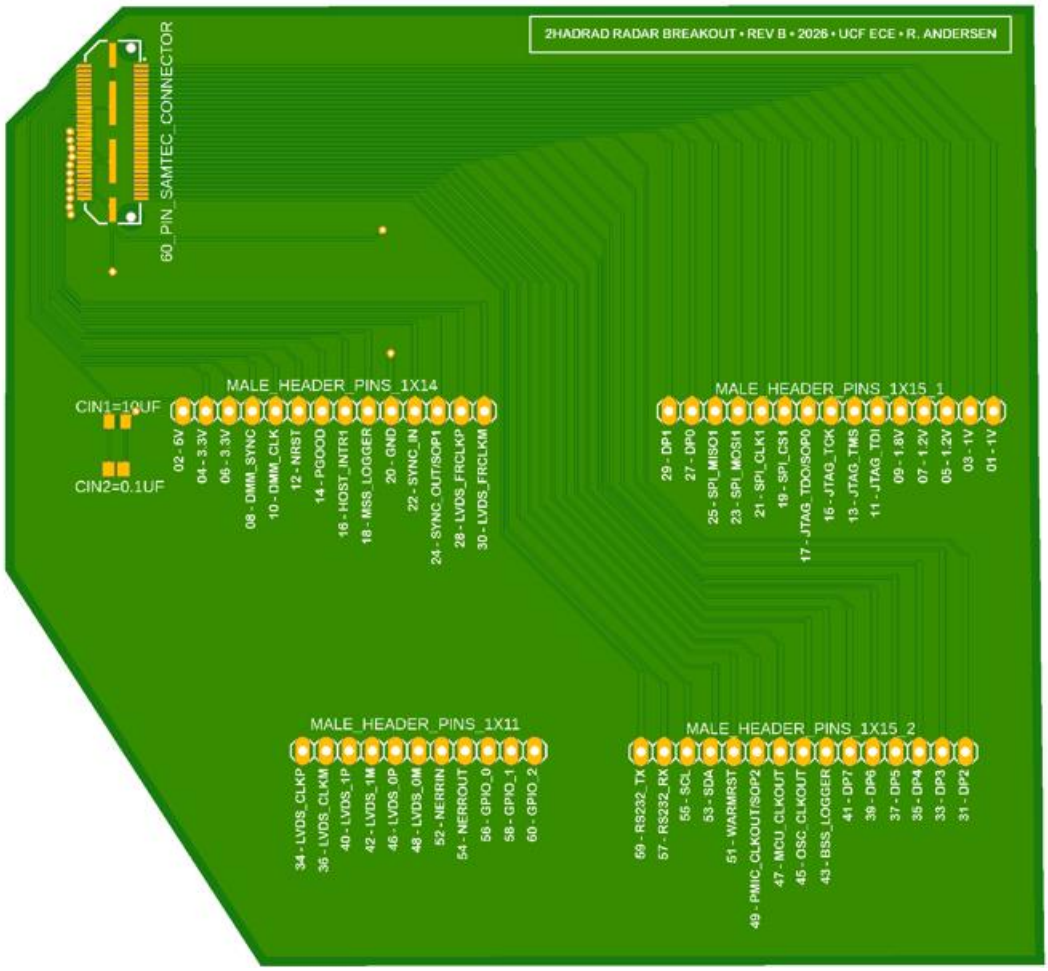


Figure 28: 60-pin Radar Breakout Board

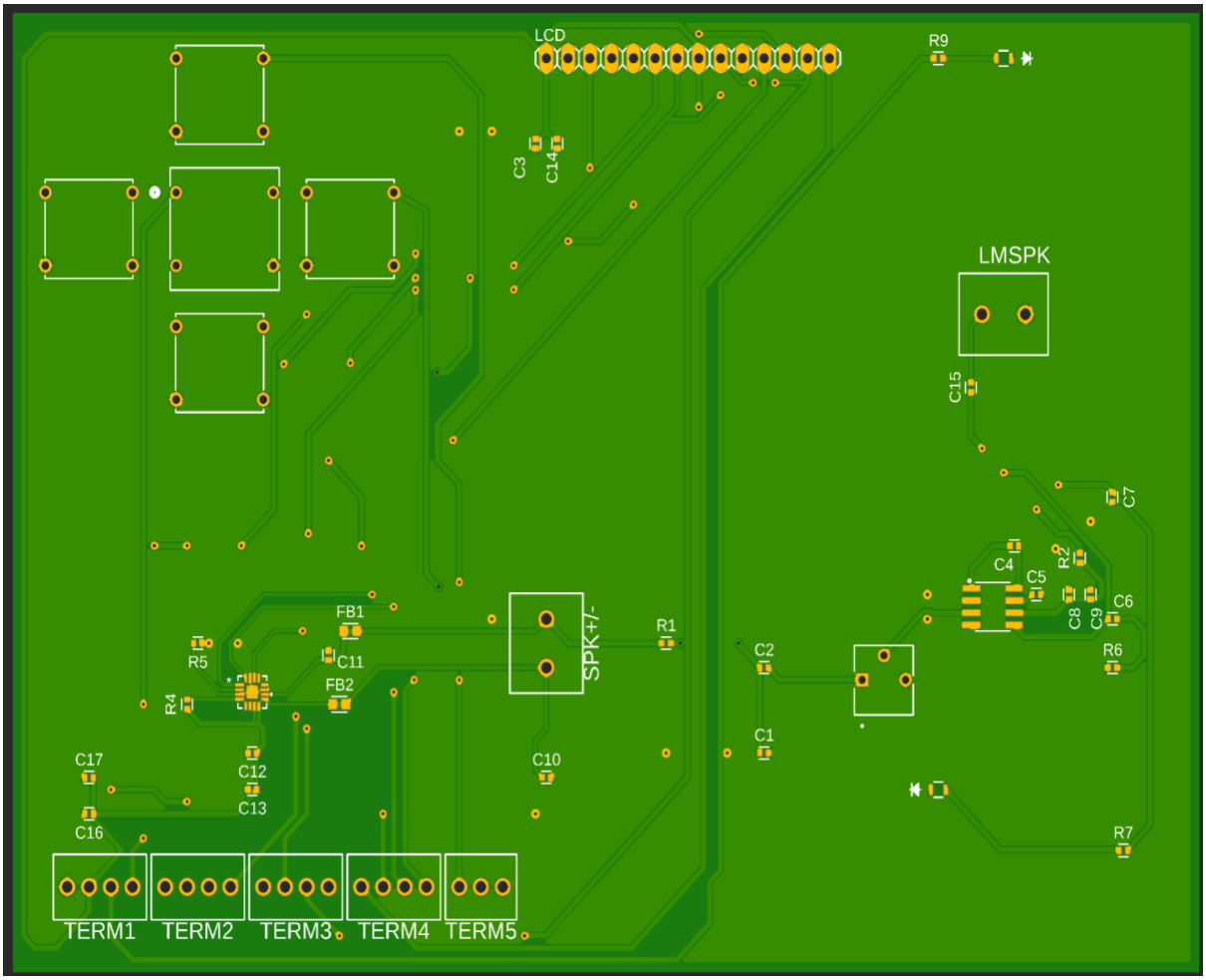


Figure 29: User Interface PCB

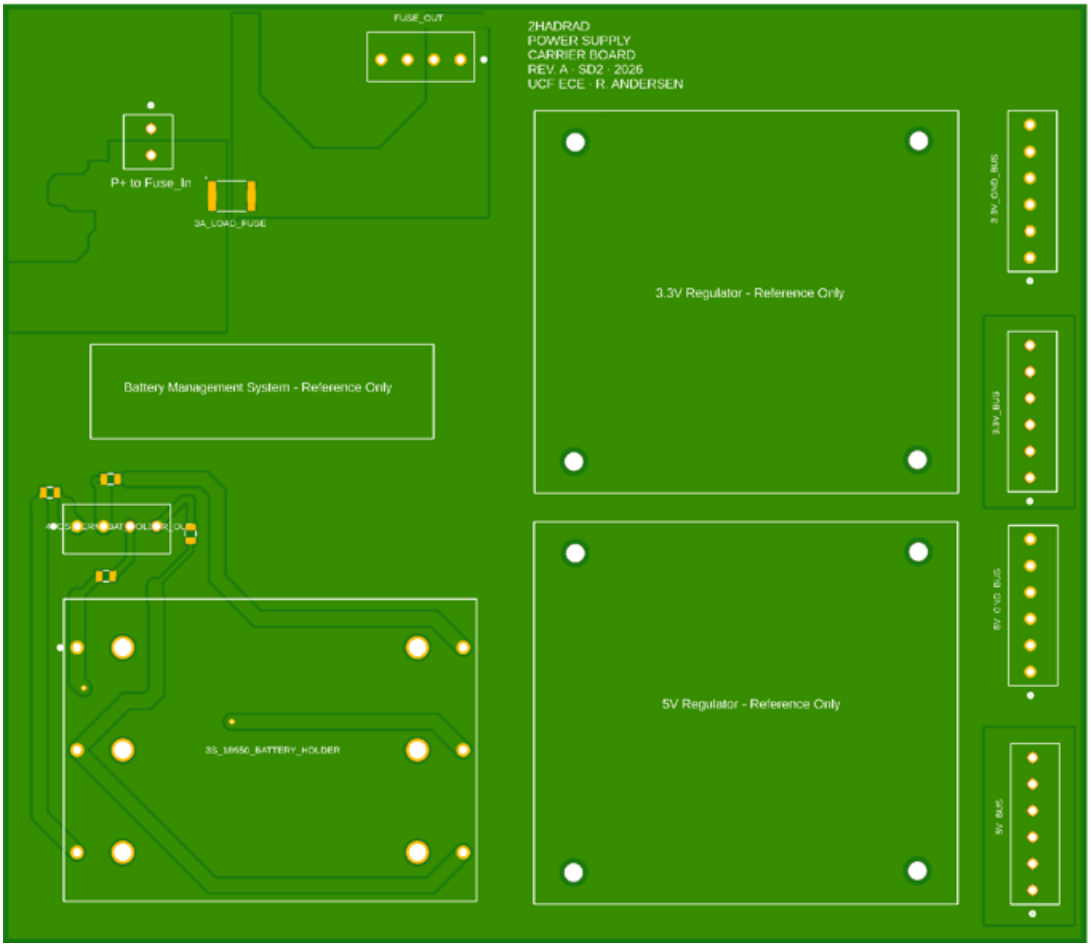


Figure 30: Power System Carrier Board

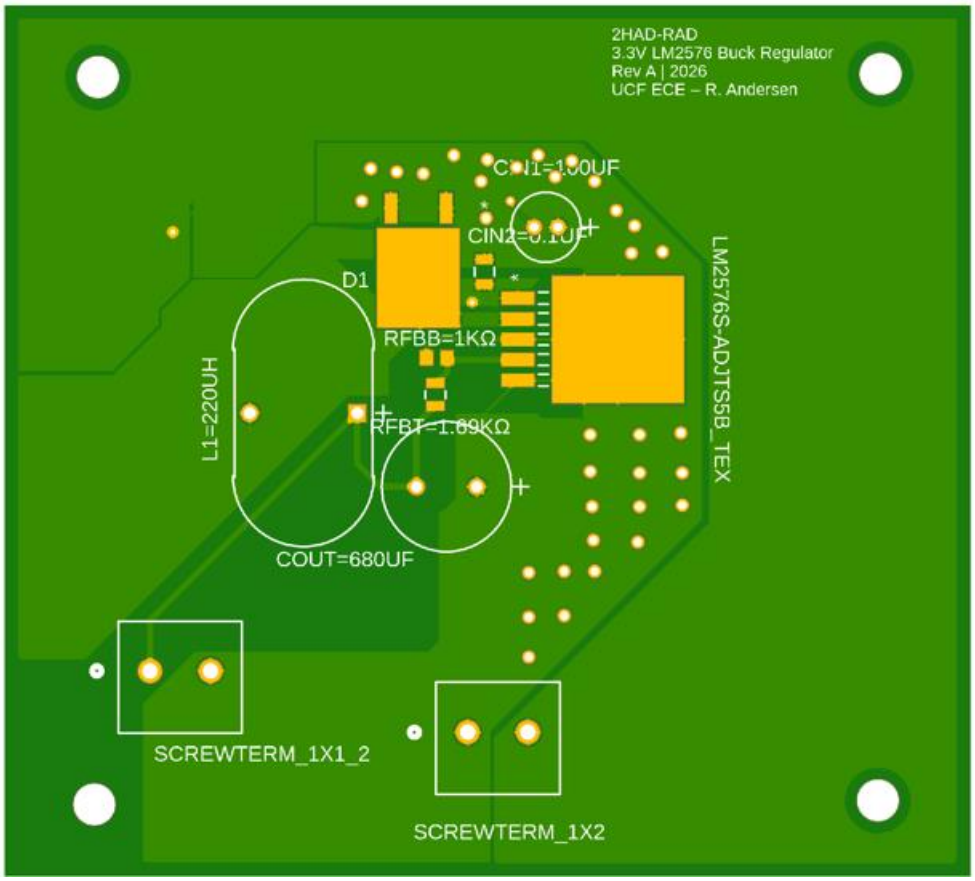


Figure 31: LM2576 3.3V Board

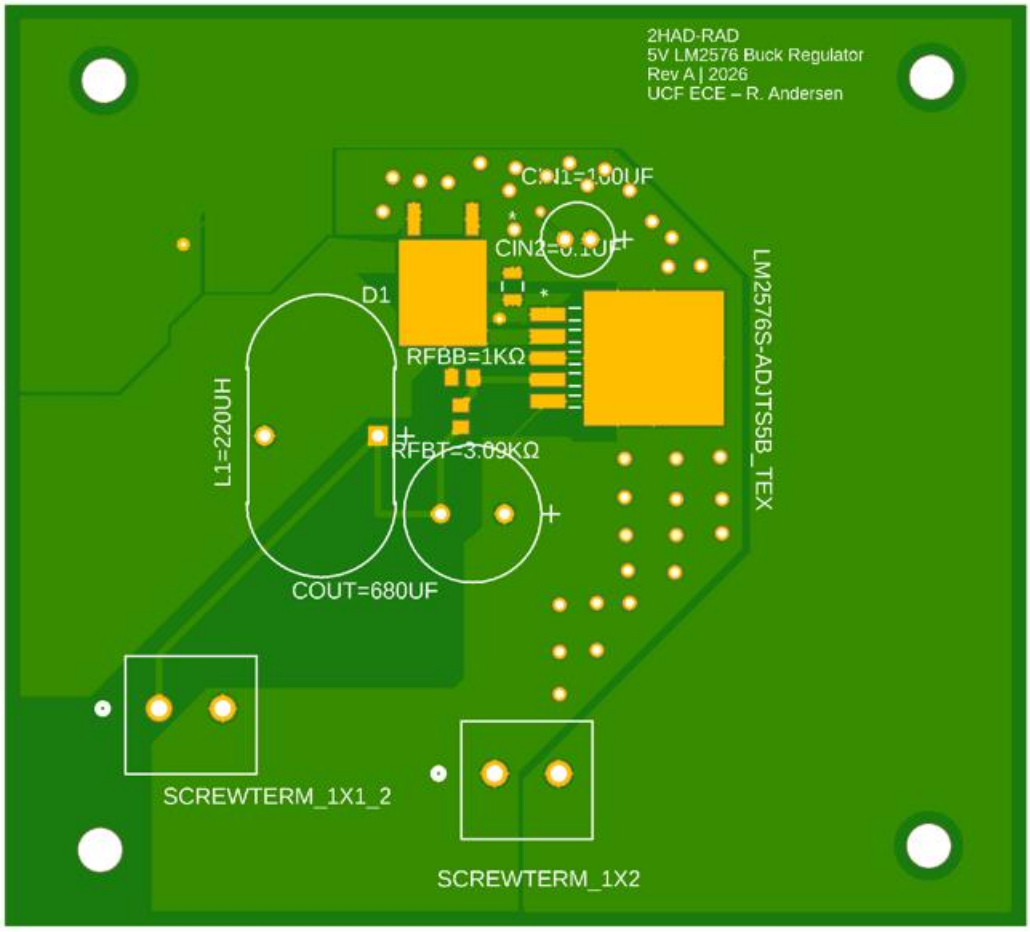


Figure 32: LM2576 5V Board

8.2. Prototype Construction

For the prototype construction, we started with a breadboard that integrated an ESP32-WROOM-32 as the main microcontroller. We first verified stable 3.3 V and 5 V rails from the onboard LDO regulator by probing the test points with a multimeter, confirming clean power delivery before populating any peripherals. The 2.8" ILI9341 TFT with XPT2046 touch controller was connected using SPI: CS on GPIO 14, DC on GPIO 27, RST on GPIO 33, and touch CS on GPIO 5 with IRQ on GPIO 4. The MLX90640 thermal camera module was wired to the I2C bus using SDA on GPIO 21 and SCL on GPIO 22, sharing the bus with the BMI270 IMU at address 0x69. For ultrasonic ranging, the HC-SR04 trigger was tied to GPIO 12 and echo to GPIO 13. The M135 (u-blox NEO-M9N) GNSS module was routed through UART2 with RX on GPIO 17 and TX on GPIO 16 after setting the module's DIP switches to CORE1 position. All peripheral power was supplied directly from the regulated 3.3 V line to minimize noise and ensure stable operation during testing. This GPIO arrangement kept all critical interfaces on dedicated hardware peripherals (SPI, I2C, UART2) while avoiding conflicts across the sensors and display.

We began by verifying the MLX90640 thermal camera as the first functional subsystem. A simple test sketch displayed a live 32x24 thermal map scaled to full-screen 320×240 resolution using a 10x10 pixel cell size and a human-optimized color palette ranging from 28 °C to 42 °C. This confirmed both I2C communication and real-time rendering on the display.

Next, we integrated the HC-SR04 ultrasonic sensor on the breadboard (Figure 30). The existing code was expanded so the thermal view and ultrasonic distance were displayed simultaneously, proving that the ESP32 could handle concurrent I2C and timed-pulse operations without timing conflicts. We tested the functionality of the IR sensor with the added USS code to ensure it operates correctly. Shown in figure 32 is the IR sensor detecting a human hand.

To create a clean UI, we implemented a touch-enabled menu system on the LCD shown in figure 32. Using the touchscreen controller, we mapped the touch coordinates and created two large buttons: one for thermal imaging and the other for proximity. Tapping either tile instantly switched modes while keeping the screen completely stable.

Finally, we developed and tested the proximity alarm feature, shown in figure 33. When the ultrasonic sensor measured a distance below 30 cm, the display immediately switched to a full-screen red warning with the text “TOO CLOSE!” in large white letters. This alarm is designed to later function as an interlock. When the distance falls below the specified threshold, the code automatically sends a sensorStop command to the radar module, thereby ensuring compliance with FCC regulations to maintain a minimum distance of 22 cm.

Each step was powered exclusively from the onboard LDO regulator’s 3.3 V rail, ensuring clean, stable power throughout the prototype phase and confirming that all chosen GPIO assignments and hardware peripherals operate reliably together on the final PCB. We began testing the radar module and the INS system, which is still a work in progress.

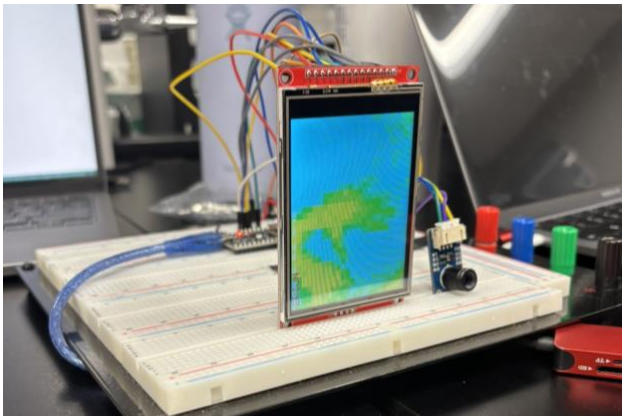


Figure 33: Testing the IR sensor with the LCD.

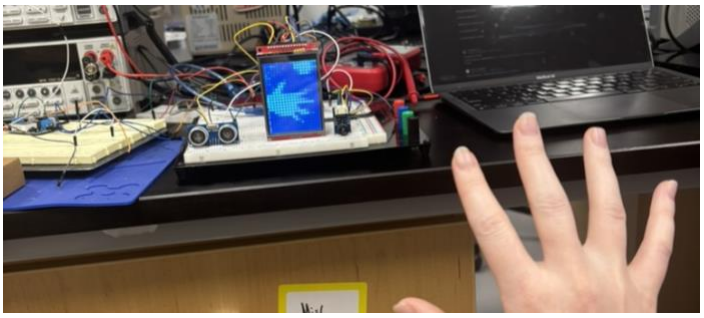


Figure 34: Additional testing done using a human body part.

Figure 35: Developed a menu to toggle between sensors.

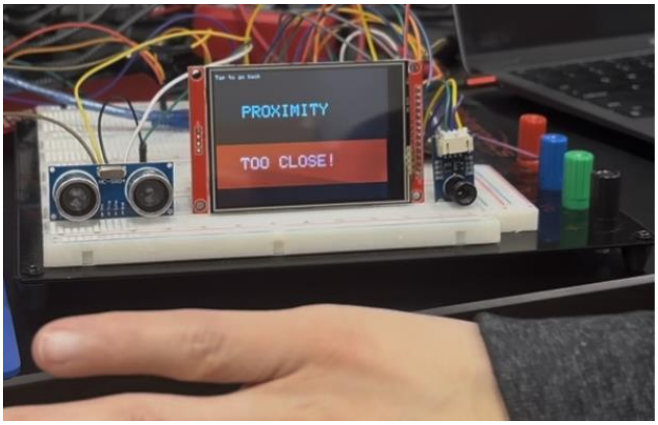


Figure 36: Testing the USS to ensure it displays when someone is too close.

8.3. Components

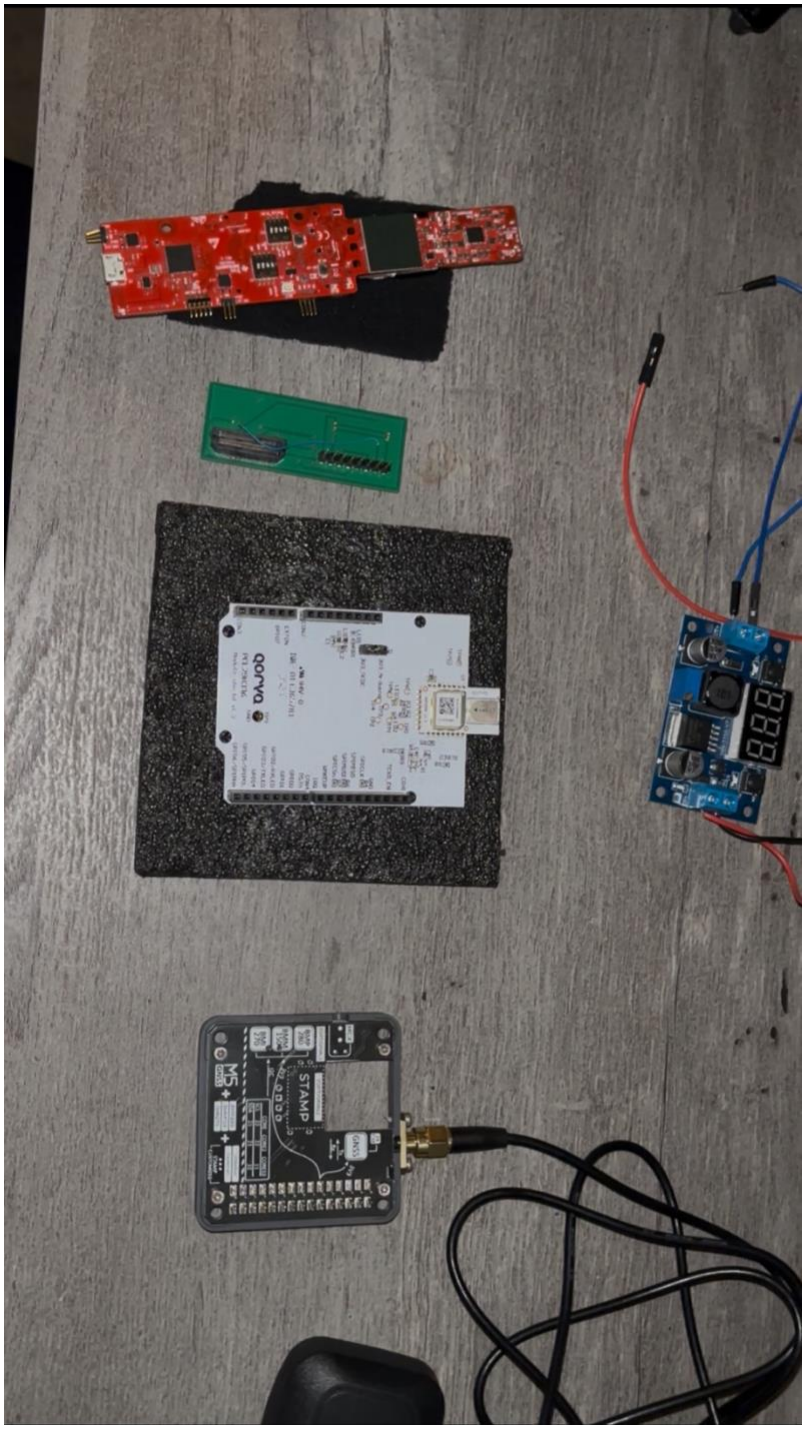


Figure 37: Components acquired aside from those listed above: Radar module, UWB sensor, INS.

CHAPTER 9

9.1. Hardware Testing

Hardware testing was done step by step as each component was added to the system. Our goal was to make sure every part worked properly on its own before connecting everything together. We used a digital multimeter throughout testing to check power lines, measure voltages, and verify that signals were reaching the correct pins. Once basic checks passed, we ran small test programs to confirm communication between the sensors and the microcontroller.

9.1.1. MCU

The ESP32-WROOM-32E microcontroller serves as the central processor for the entire system, coordinating all sensor inputs and UI outputs. To test the MCU, we first powered it through USB and verified that the onboard LED turned on, confirming that the board was active and receiving proper voltage. The regulated 3.3 V output was checked using a digital multimeter (DMM) and remained stable under load.

We tested the general-purpose I/O (GPIO) pins by connecting the ILI9341 TFT LCD and running a simple display program to confirm SPI communication. The LCD successfully displayed colors and test patterns, showing that the SPI lines (CS, DC, RST, CLK, MOSI) were functional. Similarly, the I2C pins were tested with the MLX90640 IR thermal sensor, confirming that temperature data could be read and displayed on the screen. These results verified that both communication protocols were reliable for our setup.

For future testing, once the radar module is fully connected, we will use the ESP32's secondary UART interface to communicate with the radar's microcontroller subsystem. This test will include verifying UART data transmission using serial prints, confirming radar initialization through UART responses, and ensuring synchronization with the main system loop. Overall, the MCU demonstrated stable operation, and all major communication lines functioned as intended.

9.1.2. Radar Module

The TI IWR6843AOP radar module will be one of the key sensing components in our system, responsible for detecting movement and potential human presence using millimeter-wave technology. Once interfaced with the ESP32, our focus will be verifying power stability**,** UART communication**,** and data consistency**.** Initial testing will begin by powering the module through the regulated 5 V line and confirming LED indicators to ensure proper startup. We will then connect the radar's UART pins (TX, RX, and MSS lines) to the ESP32 and use a serial terminal to monitor data output. Successful communication will be indicated by receiving initialization messages or raw radar data frames. After confirming basic

UART communication, we plan to test detection capability by placing objects at various distances and monitor the module's output to confirm range accuracy. Once we validate distance readings, we will integrate the radar data with the IR and ultrasonic sensors to perform sensor fusion tests. These will help confirm whether multiple sensing methods can improve human detection reliability under different conditions.

9.1.3. Inertial Navigation System

Initial INS testing was conducted using the M5Stack M135 module, which integrates a u-blox NEO-M9N GNSS receiver, Bosch BMI270 IMU, BMM150 magnetometer, and BMP280 barometer. Male header pins were inserted to the rear interface for jumper-wire connections, continuity was confirmed across power and communication pins with a DMM, and raw sensor data was verified over the Arduino IDE serial monitor. The accelerometer and gyroscope responded correctly to tilt and rotation, confirming I²C communication on the shared bus.

The M135 was subsequently unavailable for the final build and the INS subsystem was redesigned using discrete components: an MPU9250 9-axis IMU and a standalone BMP280 barometric pressure sensor, both connected over the same I²C bus. The MPU9250 was verified using the same approach — serial readback of raw accelerometer, gyroscope, and magnetometer registers — and the BMP280 was confirmed via pressure and temperature readback against a known reference. Quick calibration routines were then developed and validated: calibrateIMUQuick() collects 300 samples for accelerometer and gyroscope bias estimation and calibrateAltitudeBaseline() collects 12 samples for barometric altitude zeroing, both executing at power-up and on START press. Testing confirmed stable yaw initialization and acceptable short-term drift of less than 2 m over 10 m paths when the still-detection gate is active. The MPU9250 and BMP280 are the components integrated into the final firmware and PCB build.

9.1.4. Infrared Thermal Sensor

The MLX90640 IR thermal camera will provide thermal imaging for human detection. The sensor was connected to the ESP32 through I2C communication lines, and we verified proper 3.3 V power delivery with a DMM. Initial tests involved running the Adafruit MLX90640 example code to visualize temperature readings. The sensor successfully detected thermal variations between ambient objects and a human hand at short distances, showing readings in the 34–37 °C range.

In upcoming testing, we plan to record temperature data across different environments and distances to evaluate the sensor's accuracy and consistency. We will also compare the MLX90640's temperature readings to reference measurements from a digital thermometer. This will help confirm calibration and

determine emissivity correction factors for accurate human-detection use in the field.

9.1.5. Ultrasonic Sensor

The HC-SR04 ultrasonic sensor will serve as a close-range detection system to complement the radar and IR sensors. For testing, we connected the TRIG and ECHO pins to the MCU and verified voltage levels to ensure compatibility with the ESP32's 3.3 V logic. Using a test sketch, we measured the ToF of the ultrasonic pulse to calculate distance.

Preliminary tests showed accurate readings from 5 cm up to about 1 meter when compared to physical ruler measurements. In the next phase, we will repeat these tests under different environmental conditions (outdoors, varying angles, and different materials) to confirm reliability. Once integrated, the USS will be used as a trigger sensor, detecting nearby objects or people and activating other modules such as the radar for targeted sensing.

9.1.6. Power Regulation and Protection

The system uses two switching buck regulators to generate the required 5 V and 3.3 V supply rails for all components. During initial testing, the output voltages of both regulators were verified using a digital multimeter (DMM) to confirm proper regulation under load conditions. The 5 V buck converter provides power to higher-power subsystems such as the radar module and LCD backlight, while the 3.3 V buck converter supplies the ESP32-WROOM microcontroller and all low-voltage sensors and peripherals.

Because both rails are generated using switching regulators, the design prioritizes efficiency and current capability over ultra-low-noise performance. Adequate decoupling capacitors are included at the outputs to maintain voltage stability and reduce ripple during dynamic load changes.

In future testing, once the lithium-ion battery pack is fully integrated, the performance of the regulators will be evaluated under real operating conditions. Current draw will be measured using an inline ammeter to estimate total system power consumption in both active and idle states. Additionally, system behavior during load transients—such as radar activation and display updates—will be observed to ensure voltage rails remain stable.

These tests will validate that the power subsystem can reliably support portable operation while maintaining safe and efficient performance across all system modes.

9.1.7. User Interface Components (LED, Button, Speaker/Amplifier, LCD)

The UI components were tested individually before full system integration. Each pushbutton was verified for proper electrical operation using continuity mode on the DMM, and stable input readings were confirmed in the Arduino IDE serial monitor with debounce logic active. The red LED on GPIO4 was tested by toggling the pin and confirming illumination at the expected logic level. The 5-way navigation button was verified through the CAT9555 I²C GPIO expander by reading port register values in the serial monitor and confirming that each direction and the center press produced the correct bit pattern.

The MAX98357A I²S digital amplifier and speaker were tested by generating a simple tone through the ESP32 I²S peripheral and confirming audible output at the expected frequency. The software filter chain - 120 Hz high-pass, 3200 Hz three-pole low-pass, noise gate, and 0.3 gain scalar - was verified by listening for clean, hiss-free output and confirming that the amplifier muted correctly when idle. Audio playback was then tested concurrently with LCD rendering to confirm that the temporary LCD pause logic during AUDIO_PLAYING state prevented display corruption without causing visible freezing between audio events.

The ILI9341 TFT LCD was verified with color-fill and text pattern tests confirming SPI communication, then tested across all four display modes - Raw Data View, INS Map View, Radar Map View, and Thermal Heatmap View - under full concurrent sensor load. No display corruption or SPI bus conflicts were observed, confirming that the spiBusMutex arbitration and cat9555Port0Shadow register correctly coordinated access between the LCD and DW3000 UWB module on the shared SPI bus.

9.1.8. System Level Hardware Integration

System-level hardware integration testing was conducted after all individual subsystems had been verified independently. The full sensor stack - IWR6843AOP radar, MLX90640 thermal camera, MPU9250 IMU, BMP280 barometer, DW3000 UWB, HC-SR04 ultrasonic sensor, CAT9555 GPIO expander, ILI9341 LCD, and MAX98357A audio amplifier – was brought up concurrently on the ESP32-WROOM-32E under FreeRTOS.

The primary integration concern was bus contention across the shared SPI and I²C lines. The SPI mutex (spiBusMutex) was validated by running simultaneous LCD rendering and UWB ranging exchanges and confirming that no garbled frames or display artifacts appeared under sustained load. The CAT9555 shadow register (cat9555Port0Shadow) was verified by confirming that the TFT reset pin (P0.3) was never inadvertently pulled low during concurrent USS trigger (P0.1) and touch CS (P0.7) writes, eliminating the white-screen flash observed in early

bring-up. The I²C bus scheduling was verified by confirming that the MLX90640, MPU9250, BMP280, and CAT9555 all responded correctly during simultaneous polling without NACK errors or bus hangs.

The UART time-multiplexing strategy was validated end-to-end: radar configuration commands were sent over UART2 at 115200 baud during startup, the port was reassigned to GNSS after sensorStart was confirmed, and high-speed TLV data continued uninterrupted on GPIO33 at 921600 baud throughout. The 60 ms inter-command delay was confirmed to eliminate the sensorStart Error -1 fault across multiple power cycles.

The proximity interlock was tested under full system load by placing an object within 22 cm of the HC-SR04 and confirming that the radar NRST line was asserted LOW, the CLIP_TOOCLOSE audio clip fired on the same edge, and the TOO CLOSE banner displayed live USS distance. On object removal, radar revival, NRST release, and stopAudioNow were confirmed to execute correctly in sequence.

Power rail stability was confirmed under combined load using a bench supply with current monitoring. Both the 3.3 V and 5 V LM2576 rails held within specification during simultaneous radar scanning, UWB ranging, BLE advertisement, LCD updates, and audio playback - the highest concurrent load condition the system encounters in normal operation.

9.1.9. Prototype

After verifying that each hardware module functioned individually, we began assembling and testing the first version of our prototype system. This prototype combined the MCU, IR thermal sensor, ultrasonic sensor, and TFT display on a breadboard setup. The goal of this stage was to confirm that the sensors could operate together without interference and that the ESP32 could handle data collection and display updates in real time.

We connected all sensors to their designated power and communication pins based on the wiring diagram. The 3.3 V LDO regulator supplied the MCU while the 3.3 V buck regulator powered the sensors. Once powered on, the LCD successfully initialized, and we were able to view real time temperature readings from the MLX90640 while simultaneously displaying distance data from the ultrasonic sensor. These results confirmed that our wiring, power regulation, and communication buses were functioning correctly under shared operation.

During testing, we monitored voltage levels and confirmed that the system maintained steady power delivery even when multiple components were active. We also checked for overheating or voltage drops that could indicate insufficient current handling. The ESP32’s serial monitor output helped us verify data flow between modules, and we used debug printouts to ensure that sensor readings updated within a few hundred milliseconds.

The prototype provided valuable insight into how the final system will behave once fully integrated. It also allowed us to identify small layout improvements, such as organizing the jumper connections more efficiently and adding decoupling capacitors closer to the sensors to reduce noise. During this stage, we also began testing the radar module by powering it through the 5 V regulator and verifying startup indicators while preparing to establish UART communication with the MCU. In parallel, we began integrating the INS module into the prototype. By connecting it via serial communication, we observed real-time acceleration and rotation data on the serial monitor, confirming that the module was transmitting properly. During testing of the GNSS functionality, the serial monitor displayed character data such as:

Sats: 0
HDOP: 0
Chars: 60 → 370
Failed: 0
NO FIX — INDOORS OK

The changing character count indicated that the GNSS module was actively transmitting data strings, confirming proper communication with the MCU, even though satellite lock had not yet been achieved at that stage. This low character count suggests that more testing and tweaking are required within the code, as well as verifying that the switches are configured correctly. These initial integration tests helped us verify that the sensors could coexist on shared power and communication lines without interference, and they laid the groundwork for more advanced data fusion testing in the next phase.

9.1.10. Senior Design 2 Hardware Plan

The hardware integration objectives defined for Senior Design II were fully completed. The custom PCBs - main MCU/UWB/IR/INS board, radar interface breakout board, UI/display board, and power subsystem carrier board - were fabricated, assembled, and brought up successfully. Each board includes local decoupling at every supply pin and 50 Ω series resistors on high-speed SPI and UART lines.

Bring-up proceeded subsystem by subsystem as described in the testing sections above. Key integration challenges resolved during this phase included the CAT9555 read-modify-write corruption causing TFT white-screen flashes, fixed with the cat9555Port0Shadow register; the DW3000 spiSelect() library corruption inherited from DW1000-era code, fixed by reducing the function to a single reselect() call; the sensorStart Error -1 fault resolved with a 60 ms inter-command delay; and intermittent UWB ranging dropouts traced to resistive voltage drop on supply cables and eliminated by replacing them with 16 AWG conductors.

The companion anchor unit was also fabricated and integrated during this phase. The Arduino Uno R4 WiFi hosts the DWM3000EVB SS-TWR Pong responder, the Waveshare HMMD 24 GHz mmWave radar on UART, and a 16×2 I²C LCD, with all three peripherals operating concurrently on separate buses requiring no arbitration. The anchor firmware in Radar2.ino implements the full vital-sign DSP pipeline described in Section 7.1.6 and transmits breathing rate, heart rate, and UWB distance to the primary ESP32 over BLE.

The complete system was validated against the requirement specifications in Tables 1 and 2. Radar detection was confirmed at distances up to 2 m with breathing rate error improving from 10.8% at 0.5 m to 3.5% at 1.5 m. UWB ranging achieved consistent line-of-sight performance after antenna delay calibration to 16385. Thermal detection operated reliably in complete darkness and under varying ambient lighting. The 3S 18650 Li-ion pack with dual LM2576 regulators projected over five hours of runtime at the measured 3.7 W system draw, well exceeding the 2-hour requirement. Companion iOS and Android applications were confirmed operational against the same ESP32 BLE characteristic set.

9.2. Software Testing

Software testing was conducted incrementally as the firmware grew in complexity. The Arduino IDE was configured with the ESP32 board support package and all required libraries were installed, including the Adafruit MLX90640, Adafruit GFX, Adafruit ILI9341, DW3000 UWB, MPU9250, TinyGPS++, and FreeRTOS support packages. A USB-A to micro-USB cable connected the ESP32 to a development laptop for programming and serial monitor access throughout all testing stages.

Each subsystem was verified independently before integration. The ILI9341 LCD was confirmed with color-fill and text pattern tests. The MLX90640 thermal camera was verified by rendering a live 32×24 frame on the display using the Adafruit library. The HC-SR04 ultrasonic sensor was tested against a ruler reference from 5 cm to 100 cm. The MPU9250 and BMP280 were verified via raw register readback over I²C, with calibration routines tested for repeatability across multiple power cycles. The DW3000 UWB ranging state machine was tested at fixed distances of 0.5 m, 1.0 m, and 2.0 m, with the DWM3000EVB anchor on the Arduino Uno R4 WiFi acting as the Pong responder. The IWR6843AOP radar was brought up by verifying TLV frame reception on the serial monitor before any parsing logic was added, confirming both the 921600 baud data UART and 115200 baud configuration UART were functional. The CAT9555 GPIO expander was tested by toggling individual port bits and confirming correct state on a logic analyzer, then verified in-system by exercising the 5-way button and USS trigger through the expander. BLE connectivity was verified by confirming characteristic notifications were received on both the Android and iOS companion applications. Full concurrent operation of all

subsystems under FreeRTOS was then validated, confirming that the SPI mutex, I²C bus scheduling, and UART time-multiplexing strategy produced no observed conflicts or watchdog resets under sustained load.

9.2.1. MCU Software

MCU software testing began by confirming the ability to flash code onto the ESP32-WROOM-32E and exercise basic GPIO output. The ESP32 was placed on a breadboard with GPIO4 connected to a red LED through a current-limiting resistor to ground, matching the final hardware assignment for the detection status indicator. A simple sketch toggling GPIO4 on a one-second interval was uploaded via the Arduino IDE over USB and confirmed working, establishing that the toolchain, board support package, USB-to-UART bridge, and GPIO output path were all functional before any sensor code was introduced. The board includes dedicated BOOT and RESET pushbuttons: holding BOOT (IO0) low while pressing RESET (EN) manually enters the bootloader for firmware flashing, and pressing RESET alone performs a hard reset without power cycling. Both buttons were confirmed functional, and the auto-reset circuit on the EN and IO0 lines was subsequently verified to allow the Arduino IDE to enter bootloader mode and complete programming automatically without requiring manual button intervention during normal development.

Subsequent MCU software testing verified each communication bus independently. SPI was confirmed via ILI9341 color-fill and text tests. I²C was confirmed via MLX90640 device address readback and live temperature streaming. All three hardware UARTs were verified: UART0 for debug output, UART2 for radar configuration at 115200 baud and GNSS reassignment at 9600 baud, and UART1 receive-only on GPIO33 for TLV data at 921600 baud. The CAT9555 GPIO expander was tested by toggling individual port bits and confirming correct output state, then exercised in-system through the 5-way button and USS trigger. FreeRTOS task scheduling was validated under full concurrent sensor load with no watchdog resets or observed scheduling conflicts across the radar, thermal, INS, UWB, display, and audio tasks.

9.2.2. UI Software

UI Software testing began with a simple sketch displaying a black background and a text string to verify SPI communication with the ILI9341 controller. The Adafruit ILI9341 library was installed and the display was wired with CS on GPIO25, DC on GPIO32, RST tied to the CAT9555 TFT reset line (P0.3), SCLK on GPIO18, and MOSI on GPIO23. The library initialized the display correctly on first attempt and color-fill tests across the full 320×240 framebuffer confirmed no dead pixels or rendering artifacts. Text rendering, colored rectangles, and gradient fills verified that the full Adafruit GFX drawing API was stable.

The dirty-rectangle caching strategy was implemented and verified by confirming that only changed screen regions triggered SPI transactions, reducing bus load during concurrent sensor operation. All four display modes — Raw Data View, INS Map View, Radar Map View, and Thermal Heatmap View — were tested under full concurrent sensor load with no display corruption or visible tearing observed. The lcdBeginAccess() and lcdEndAccess() mutex wrappers were confirmed to correctly sequence the CAT9555 I²C chip-select assertion before releasing the SPI bus, and the cat9555Port0Shadow register was confirmed to eliminate the white-screen flash caused by read-modify-write corruption on the CAT9555 PORT0 register observed during early bring-up.

The XPT2046 touch controller was wired with its CS routed through the CAT9555 expander (P0.7) and IRQ on GPIO34. Touch coordinate mapping was verified by tapping the screen and confirming that raw X/Y values translated correctly to display coordinates. A touch-enabled two-tile menu was implemented to switch between thermal imaging and proximity display modes, confirming that the CAT9555-routed touch CS and the native ESP32 touch IRQ pin worked correctly together.

The 5-way navigation button was tested through the CAT9555 I²C GPIO expander by reading port register values in the serial monitor and confirming that each cardinal direction and the center press produced the correct bit pattern. Software debounce logic using millisecond-level timing was verified by pressing each direction rapidly and confirming no false triggers were registered. Button-driven menu navigation was then tested across all four display modes, confirming that mode switching, START, MARK, and CLOSE actions all executed correctly from the button without requiring touch input. The toggle power switch was verified to correctly control system power independently of the navigation button, confirming the operator cannot accidentally power down the device during field operation.

The MAX98357A I²S digital amplifier and speaker were tested by generating a simple tone through the ESP32 I²S peripheral and confirming audible output at the expected frequency. The software filter chain — single-pole 120 Hz high-pass filter, three-pole 3200 Hz low-pass filter, noise gate at 0.005 normalized amplitude, and 0.3 output gain scalar — was verified by listening for clean, hiss-free output and confirming that the amplifier muted correctly when idle. Distinct audio clips for session start and end, human detection, proximity alarm, and UI error events were each tested individually and confirmed to trigger on the correct system edges.

Audio playback was then tested concurrently with LCD rendering to confirm that the AUDIO_PLAYING guard correctly paused LCD updates during active playback without causing visible freezing between audio events. An earlier implementation that held the guard across AUDIO_STARTING and AUDIO_FINISHING states was confirmed to cause a full screen redraw on every

audio completion, producing visible glitches; tightening the guard to AUDIO_PLAYING state alone eliminated this. The proximity interlock audio fusion was verified by triggering the USS threshold and confirming that startClip(CLIP_TOOCLOSE) fired on the same edge as radarKillHard(), that the clip looped continuously while radarKilledByProx remained set, and that stopAudioNow() fired correctly on the clear edge when the obstacle was removed.

9.2.3. Inertial Navigation System Software

INS software testing began with raw I²C register readback from the MPU9250 and BMP280 over the shared bus (GPIO21 SDA, GPIO22 SCL). A simple sketch read the MPU9250 WHO_AM_I register to confirm the device was addressable, then streamed raw accelerometer, gyroscope, and magnetometer values to the Arduino IDE serial monitor. The module was tilted and rotated in each axis to confirm that all three accelerometer axes and all three gyroscope axes responded correctly with appropriate sign and magnitude. The BMP280 was verified by reading pressure and temperature registers and confirming values were within expected range for the lab environment.

Calibration routine testing followed. The calibrateIMUQuick() function was executed and the collected 300-sample bias averages were printed to the serial monitor and confirmed to be stable across multiple power cycles. The calibrateAltitudeBaseline() function was verified by running it at a known elevation and confirming the zeroed altitude output read near zero immediately after calibration. The Madgwick filter integration was then tested by streaming quaternion and Euler angle outputs while rotating the device, confirming that roll and pitch tracked physical orientation with low noise and that yaw integration via raw gyro-Z was stable over short intervals. The still-detection gate was verified by confirming that dead-reckoning velocity zeroed correctly during stationary periods and began integrating again on deliberate movement.

9.2.4. Infrared Thermal Sensor Software

MLX90640 software testing began with the Adafruit MLX90640 library example sketch to confirm I²C communication and raw frame readback. The readFrame() function was verified to populate the 768-element pixels array with calibrated temperature values, and the serial monitor was used to confirm readings were within expected range for the lab environment before any display rendering was added.

Display rendering was then added incrementally. Each of the 768 temperature values was mapped to a color using a dynamic palette anchored to the scene ambient temperature, with cooler values rendering blue, mid-range green and cyan, warm values yellow and orange, and the hottest pixels red. Each pixel was drawn as a scaled rectangle block on the ILI9341 display. This was subsequently

replaced by the bilinear interpolation upscaling pipeline that expands the 32×24 frame to 64×48 pixels before rendering, improving visual significantly.

The fixed-threshold human detection approach — flagging clusters of more than 35 pixels in the 27–38°C range with a bounding box — was implemented and tested as an early baseline. The sensor successfully detected thermal contrast between ambient objects and a human hand at short range, with readings in the 34–37°C range confirmed on the display. This threshold-based approach was subsequently replaced by the full multi-stage classification pipeline in thermal_human_detector.h, implementing adaptive dual time-constant background modeling with BMP280 baro-temperature fusion, dynamic percentile-based thresholding, 8-connected blob segmentation, and 9-class shape scoring. The advanced pipeline eliminated the false positives from background heat sources that the fixed-threshold approach produced under varying ambient conditions.

9.2.5. Radar Software

Radar software testing began by verifying TLV frame reception on the serial monitor before any parsing logic was added. The IWR6843AOP was powered from the 5 V rail and both UART interfaces were connected: UART2 (GPIO16/GPIO17) for configuration commands at 115200 baud and UART1 receive-only (GPIO33) for high-speed TLV data at 921600 baud. The radar was programmed using TI UniFlash with the vital_signs_tracking_6843AOP_demo.bin firmware from the mmWave Industrial Toolbox v4.12.1, and runtime configuration was applied through the CLI UART using a modified vital_signs_AOP_6m.cfg profile.

Initial bring-up revealed that sending configuration commands too quickly caused a sensorStart Error -1 fault because the radar's internal ARM core had not finished processing each preceding command. A 60 ms inter-command delay on UART2 resolved this completely and the fault was not observed again across subsequent power cycles. The DIP switch configuration for correct 60-pin Samtec connector operation was determined empirically, as the configuration specified in the datasheet did not produce functional UART communication. The working configuration (S1: ON ON OFF OFF, S2: OFF ON OFF OFF, S3: OFF) was confirmed by observing the radar boot prompt on the serial monitor.

TLV frame parsing was then implemented and verified. TLV type 1021 presence flags, type 1040 vital-sign frames, and type 1010 target position frames were all confirmed to parse correctly, with HR and BR values extracted from byte offsets 8 and 12 respectively within the 136-byte type 1040 payload. Breathing rate values within the 5–40 BPM acceptance window were confirmed directionally consistent with actual respiration. Heart rate estimates were observed to cluster near 50.2 BPM regardless of true cardiac rate, consistent with the FFT subharmonic locking behavior documented in the results section, and HR was

accordingly treated as a presence-corroborating signal rather than a standalone measurement.

The two-condition detection gate requiring both targetPresent and vitalsReady was verified in an empty-room control condition, confirming that TLV 1021 presence flags triggered by background reflections did not assert humanDetected when vitalsReady was not simultaneously true, producing zero false-positive detection events. The two-phase chirp strategy was tested across multiple power cycles, confirming that the Phase 1 to Phase 2 transition executed correctly - hardware resetting the radar, waiting for the boot prompt, transmitting the Phase 2 configuration, and resuming TLV data collection on UART1 - in all tested runs.

9.2.6. BLE Software

BLE software testing began by verifying that the ESP32 advertised correctly as a peripheral and was discoverable by both Android and iOS devices. The ESP32 BLE GATT server was configured with the service and characteristic UUIDs matching the Android application's expected set, and characteristic notification was enabled. Initial testing confirmed that the Android application connected successfully, discovered services, and received characteristic notifications without requiring manual CCCD descriptor writes beyond what the ESP32 stack handled automatically.

String payload characteristics carrying radar vitals, INS state, and scan mode results were verified by confirming that data appeared correctly formatted in the Android application's live tile display with latency consistently under the 10-second requirement specified in Table 2. The raw byte thermal characteristic was verified by confirming that the 32×24 pixel array transmitted correctly and rendered as a recognizable thermal heatmap in both the Android and iOS applications.

The iOS application was verified separately by confirming that the BLEManager class connected to the ESP32 peripheral by device name, completed MTU negotiation, discovered services, and subscribed to all characteristics via setNotifyValue without requiring manual CCCD writes. Incoming data was confirmed to dispatch correctly by UUID to all five views - radar and proximity, thermal heatmap, INS navigation, scan mode results, and connection management - updating in real time without manual refresh.

BLE transmission was then tested under full concurrent sensor load to confirm that the BLE radio did not interfere with simultaneous radar UART parsing, UWB SPI ranging, I²C thermal reads, and LCD SPI rendering. No observed packet loss, connection drops, or scheduling conflicts were recorded during sustained concurrent operation. Victim snapshot transmission was verified by marking a detection, confirming the fused position, thermal class, radar vitals, and timestamp were all received correctly on both platforms, and confirming that the

connection status indicator in the persistent top bar updated correctly on link establishment and loss.

9.2.7. Senior Design 2 Software Plan

All primary software objectives defined for Senior Design II were completed. The firmware implements four fully functional display modes (Raw Data View, INS Map View, Radar Map View, and Thermal Heatmap View), a multi-phase scan mode engine, the 8-state EKF fusing IMU, UWB, and barometer, the adaptive thermal classification pipeline, the anchor unit DSP firmware in Radar2.ino, and companion iOS and Android applications confirmed operational against the same ESP32 BLE characteristic set. Senior Design II documentation, final demonstration, and recommendations for future improvements are addressed in the conclusion.

CHAPTER 10

10.1. Budget

Since this project is entirely self-funded, maintaining affordability without compromising functionality was a primary consideration throughout the component selection process. The primary objective was to maintain the total cost close to $500, ensuring a harmonious balance between performance, sensor accuracy, and system reliability. The total cost of the current bill of materials is approximately $505, which encompasses all major modules, including the radar, INS, UWB transceiver, LCD display, and supporting power management circuitry. This estimate includes an estimated cost for the PCB fabrication and assembly, which will be updated once the final design is confirmed. The team prioritized using cost-effective yet proven components, such as the ESP32 microcontroller and off-the-shelf sensor modules, to minimize prototyping expenses while ensuring future expandability. Considering the intricate nature of the system and the specialized design of the radar and thermal imaging sensors, maintaining this proximity to the target budget showcases efficient component selection and effective resource management.

Sub-System	Budget
Sensing Modules (Radar, UWB, INS, IR Thermal Sensor)	$396.80
Processing & Control (MCU, Power Regulation, MOSFETs, Inductors)	$21.00
UI (LCD, LED, Navigation Button, Amplifier/Speaker, Light Pipes)	$46.00
Power & Protection (Battery, Charger/BMS, USB, TVS, Diodes, Connectors)	$16.00
PCB Fabrication & Assembly (Estimated)	$50.00
Passive Components (Resistors, Capacitors, Misc. Hardware)	$10.00

Sub-System	Budget
Total Estimated Cost	$539.00

Table 27: Budget allocation table for sub-systems.

The table above outlines the budget according to each subsystem, providing a clear distribution of how project funds were allocated. The sensing modules, which encompass the radar, UWB, INS, and thermal imaging sensor, constitute the largest portion of the budget. These components are crucial for the device's primary life-detection function. The UI and processing systems follow, encompassing the LCD display, pushbuttons, LEDs, and the ESP32 microcontroller that coordinates all sensor operations. Power management and protection circuits, including the battery, voltage regulators, and safety components, were designed to remain efficient and affordable. Passive elements such as capacitors, resistors, and inductors add minimal cost but are essential for stable and low noise performance. Overall, this breakdown showcases a well-rounded approach that strikes a harmonious balance between functionality, durability, and cost effectiveness, all while adhering to the self-funded constraints of the project.

Component	Model	Price	Quantity
Radar Module	TI IWR6843AOP EVM	$224.35	1
INS	M5Stack M135	$50.00	1
GPS/GNSS	ATGM336H-5N	$10.69	1
IMU	MPU9250 + GY-9250	$15.59	1
Barometer	BMP280-3.3	$7.39	1
UWB	DWM3000EVB	$42.50	1
UWB	DWM3000TR13	$23.62	2
MCU	ESP32-WROOM-32E	$11.72	1
LCD	TFT ILI9488 SPI 3.2 inches	$16.99	2
Speaker	4Ω 3W Loudspeaker	$8.72	1
AMP	MAX98357A I2S 3W	$9.49	2
USB-A	USB connector	$2.34	2
Red LED	LS R976-NR-1-0-20-R18	$0.15	1
Button Sealing Support	1ZY	$1.99	1

Component	Model	Price	Quantity
Button Sealing Washer	1ZZ09	$1.01	1
Button Cap	1ZW0913611806	$8.29	1
Center Button Switch	5GTH935	$3.22	1
Other Button Switches	5ETH935	$2.55	4
IR Thermal Sensor	Adafruit MLX90640	$79.95	1
Radar Breakout Board	Custom PCB	$88.62	1
UI PCB	Custom PCB	$15.80	1
Sensor Subsystem PCB	Custom PCB	$20.50	1
Power Carrier Board	Custom PCB	$25.20	1
3.3V Reg Board	Custom PCB	$11.06	1
5 V Reg Board	Custom PCB	$9.06	1
Battery	18650 Lithium-ion	$9.00	5
Light Pipe	515-1020-805F	$3.00	1
3.3V Regulator	TPS563200	$1.10	1
5V Regulator	TPS61236P	$2.40	1
PETG 3D Material	3D- printing filament	$70.00	1

Component	Model	Price	Quantity
CAT9555 GPIO Expander	CAT9555YI-T2	$1.36	2
USB-UART	CP2102	$8.80	1
1S BMS Module	DW01A + FS8205A-based board	$1.50	1
Inductor (5V Boost)	Bourns SRP4020TA-2R2M	$1.85	1
Inductor (3.3V)	Bourns SRP5030TA-4R7M	$1.95	1
Ardunio	Ardunio UNO R4 WIFI	$23.54	1
24 GHz mmWave Radar	Waveshare HMMD-mmWave-radar	$10.99	1
Battery Storage	18650 Battery Storage case	$6.99	2
Capacitors	Assorted	$3.50	25
Resistors	Assorted	$3.00	25
Total		$839.75	

Table 28: Bill of Materials

10.2. Division of Work

The table below provides a clear breakdown of the responsibilities assigned to each member of the 2HADRAD development team. It outlines their primary and secondary roles within the project's various subsystems. This structure ensures a balanced distribution of workloads and fosters cross-disciplinary collaboration between electrical and computer engineering disciplines. Each subsystem was assigned a primary lead responsible for its design, implementation, and testing. A secondary team member provided support for verification, integration, and documentation. This collaborative approach enables the sharing of overlapping expertise, particularly in power management, sensing, and data acquisition. This ensures the system's reliability and continuity throughout the design, prototyping, and testing phases.

Subsystem	Primary Lead	Secondary Support
Power Subsystem	Ryan Andersen (EE) – Designed and integrated the Li-ion battery system, BMS, and DC/DC regulation circuits for 3.3 V and 5 V rails.	Bailey Hitt (EE) – Assisted with component sourcing, protection circuitry, and power efficiency testing.
Sensors Subsystem	Shannon Hankinson (EE) – Lead integration of UWB, INS, IR sensors, and USS managing I2C/SPI and UART communication with the MCU.	Bailey Hitt (EE) – Supported sensor calibration, system validation under field conditions, and debugging.
Radar Module	Ryan Andersen (EE) – Implements the TI IWR6843AOP radar, handling configuration, mounting, and communication with the main controller.	Shannon Hankinson (EE) – Provided firmware assistance and performance tuning for radar data fusion.
MCU	Shannon Hankinson (EE) – Developed firmware, SPI/UART interfacing, and data acquisition timing control.	Bailey Hitt (EE) and Joshua D'Souza (CpE) – Supported hardware verification and debugging of MCU-based communication and supports firmware development.
UI	Bailey Hitt (EE) – Designed and codeed LCD interactions, LED logic, button handling, and speaker control.	Shannon Hankinson (EE) and Joshua D'Souza (CpE) – Assist with signal routing, interface layout, UI testing, and coding.

Table 29: Division of Work

10.3. Milestones

Milestone	Start Date	Anticipated End Date
Project Brainstorming	8/19/25	8/26/25
Project Scope Finalized	8/26/25	9/6/25
Individual Research	8/26/25	9/6/25
Initial Proposal Submission	8/26/25	9/6/25
30-Page Milestone	9/20/25	9/27/25
60-Page Milestone	9/6/25	9/27/25
80-Page Milestone	9/27/25	10/31/25
90-Page Milestone	10/31/25	11/7/25
120-Page Milestone	11/7/25	11/25/25
Final Draft	11/25/25	12/3/25

Table 30: Senior Design 1 documentation milestones

Milestone	Start Date	Anticipated End Date
System Design	11/1/25	11/15/25
System Testing	11/16/25	11/22/25
Prototype	11/23/25	11/29/25
PCB	11/20/25	12/6/25

Table 31: Senior Design 1 design/prototype milestones

Milestone	Start Date	Anticipated End Date
PCB Testing	1/12/26	1/31/26
System Integration and Testing	2/1/26	3/14/26
Practice Demo	3/15/26	3/28/26
Finalize Documentation	3/29/26	4/28/26

Milestone	Start Date	Anticipated End Date
Practice Final Presentation	4/12/26	4/21/26
Final Presentation	4/21/26	4/21/26

Table 32: Estimated Senior Design 2 milestones

The overall project plan outlines a well-defined timeline for the development of the 2HADRAD system, providing a roadmap for the team from the initial concept phase to the complete integration and testing of the system. The milestones are strategically spaced to provide sufficient time for research, design validation, and hardware development, ensuring clear deliverables at each stage. While the Senior Design 1 schedule offers a comprehensive roadmap, including detailed documentation and preliminary prototyping, the timeline for Senior Design 2 is still an initial estimate. As the project progresses and more design details are finalized, this schedule will be revisited and refined to reflect the actual progress, component lead times, and testing outcomes. This ensures that the completion plan is realistic and achievable.

CHAPTER 11

11.1. Conclusion

The development of the 2HADRAD system represents a substantial achievement in creating a compact, field-deployable life-detection device capable of supporting first responders in hazardous and time-critical search and rescue environments. Throughout this project, the team investigated, analyzed, and integrated a wide range of sensing technologies - millimeter-wave radar, UWB ranging, IR thermography, inertial navigation, barometric pressure sensing, GNSS, and supporting UI and power subsystems - into a handheld platform designed specifically for real-world search and rescue needs. The resulting system architecture reflects a synthesis of modern sensing hardware, embedded signal processing, and practical design constraints that collectively moved the device from concept to functional deployment.

A core objective of 2HADRAD is providing reliable detection of human presence in conditions where traditional methods such as visual inspection or thermal imaging alone may fail. The IWR6843AOP radar was selected as the primary sensing modality due to its ability to detect small, periodic micro-motions - such as breathing - even in the presence of dust, smoke, or reduced lighting. By integrating IR thermal sensing through the MLX90640, UWB ranging through the DW3000 and DWM3000EVB anchor, and inertial motion data from the MPU9250 and BMP280 through the 8-state Extended Kalman Filter, the system strengthens detection reliability through multi-modal confirmation. This approach

significantly reduces false positives, provides higher detection confidence, and offers spatial context that single-sensor systems cannot achieve.

Localization of victims was addressed through the INS dead-reckoning pipeline, Madgwick AHRS filter, EKF fusion, and GNSS integration via TinyGPS++ over time-multiplexed UART. Indoors or underground, the INS maintains motion tracking autonomously; when GNSS is available, absolute coordinates reinforce accuracy and correct drift. The BMP280 introduces a vertical dimension, enabling responders to distinguish floor-level changes in multi-story or collapsed structures. This layered localization approach directly supports search and rescue efforts by providing fused victim coordinates even in GPS-denied environments.

The hardware and software subsystems developed for 2HADRAD reflect an intentional focus on robustness and usability in the field. Firmware executes filtering, signal conditioning, DSP algorithms, and sensor fusion logic in real time under FreeRTOS on the ESP32-WROOM-32E, while the UI provides immediate and intuitive feedback under difficult environmental conditions across four display modes - Raw Data View, INS Map View, Radar Map View, and Thermal Heatmap View. The MAX98357A I²S audio amplifier, 5-way navigation button through the CAT9555 GPIO expander, and ILI9341 TFT display with XPT2046 touch extend usability in high-noise and low-visibility environments. The companion iOS and Android applications extend situational awareness to rescue teams over BLE, providing real-time victim snapshots, thermal frames, radar vitals, and fused sensor readings on both platforms. The anchor unit on the Arduino Uno R4 WiFi contributes an independent 24 GHz mmWave vital-sign extraction pipeline and SS-TWR Pong responder, providing through-wall sensing capability from a fixed vantage point where the handheld primary device cannot maintain stable pointing.

The power subsystem, built around a 3S 18650 Li-ion pack with dual LM2576 buck regulators, delivered a measured system draw of approximately 3.7 W under full concurrent sensor load, projecting over five hours of runtime - well exceeding the two-hour design requirement. Bus arbitration through the spiBusMutex and cat9555Port0Shadow register, UART time-multiplexing across radar configuration and GNSS, and FreeRTOS task scheduling across all concurrent subsystems were all validated under sustained load with no watchdog resets or observed scheduling conflicts.

The complete system was validated against the requirement specifications in Tables 1 and 2. Radar detection was confirmed at distances up to 2 m with breathing rate error improving from 10.8% at 0.5 m to 3.5% at 1.5 m. UWB ranging achieved consistent line-of-sight performance after antenna delay calibration to 16385. Thermal detection operated reliably in complete darkness and under varying ambient lighting. BLE transmission to both iOS and Android

platforms were confirmed within the 10-second latency requirement under full concurrent sensor load.

Looking ahead, the modular architecture of 2HADRAD provides a clear pathway for future enhancement. Promising directions include expanded ML-assisted thermal and radar classification, integration of seismic sensing for through-rubble detection, improved UWB multi-anchor positioning using additional DWM3000EVB nodes, and battery monitoring with runtime estimation displayed on the UI. The scan mode engine and EKF fusion layer are intentionally designed to accommodate additional sensor modalities without requiring architectural changes to the existing firmware. As the device approaches further refinement, its potential applications extend beyond disaster response to include military personnel tracking, industrial safety monitoring, and security screening in GPS-denied environments.

Ultimately, the work completed by the 2HADRAD team demonstrates that a lightweight, multi-sensor, embedded life-detection platform is both technically feasible and practically valuable. By combining 60 GHz FMCW radar, IR thermal classification, UWB ranging, inertial navigation, and user-centered interface design into a single handheld device with companion mobile applications, 2HADRAD advances the state of portable survivor detection tools and contributes meaningful engineering innovation to the broader field of emergency response technology. Every design decision - from the SS-TWR ranging architecture and the adaptive thermal classification pipeline to the bus arbitration strategy and the audio interlock - was made in service of a single goal: helping first responders find survivors faster in environments where every second matters.

Appendix A - References

[1] O. B. Lubecke, P.-W. Ong, and V. M. Lubecke, “10 GHz Doppler radar sensing of respiration and heart movement,” IEEE Xplore, [Online]. Available: https://ieeexplore.ieee.org/document/999462

[2] Infineon Technologies AG, BGT60TR13C Radar Sensor Datasheet, Rev. 1.0, 2020. [Online]. Available: https://www.infineon.com/dgdl/Infineon-BGT60TR13C-DS-v01_00-EN.pdf

[3] Federal Communications Commission, Title 47 CFR Part 15 – Radio Frequency Devices, U.S. Government Publishing Office, 2023. [Online]. Available: https://www.ecfr.gov/current/title-47/chapter-I/subchapter-A/part-15

[4] International Electrotechnical Commission, IEC 62133-2: Secondary Cells and Batteries… for Portable Applications, 2nd ed., Geneva, Switzerland, 2017.

[5] Underwriters Laboratories, UL 2054: Household and Commercial Batteries, Northbrook, IL, USA, 2011.

[6] IEEE Standards Association, IEEE 802.15.4™-2020 – Low-Rate Wireless Personal Area Networks (LR-WPANs), IEEE, New York, NY, USA, 2020.

[7] Department of Homeland Security, Portable Signs of Life Identifier: Technology Scouting Research Summary, Aug. 2019. [Online]. Available: https://www.dhs.gov/sites/default/files/publications/portable_signs_of_life_updated-508c_v3.pdf

[8] Department of Homeland Security, Detection of Presence of Life Through Walls, S&T Directorate, Dec. 2022. [Online]. Available: https://www.dhs.gov/sites/default/files/2022-12/22_1214_st_Detection_Presence_Life%20throughWalls_122022.pdf

[9] G. Charvat, Small and Short-Range Radar Systems, CRC Press, 2014.

[10] H. Mostafa et al., “Sensor Fusion for Non-Contact Human Presence Detection,” IEEE Sensors Journal, vol. 21, no. 10, pp. 11938–11947, 2021.

[11] Camero-Tech Ltd., Xaver™ 100 and 400 Product Overview, 2023.

[12] Vayyar Imaging, Vayyar Sensor Technology Overview, Technical Datasheet, 2023.

[13] S. Wang et al., “Combined UWB Radar and Infrared Sensing for Human Vital Sign Detection,” Sensors, vol. 23, no. 5, 2023.

[14] A. Kharbouch et al., “Hybrid Radar–Thermal Sensor System for Disaster Victim Localization,” IEEE Trans. Geosci. Remote Sens., vol. 60, pp. 1–14, 2022.

[15] Texas Instruments, IWR6843AOP Single-Chip 60–64 GHz mmWave Sensor (AoP) – Datasheet, Rev. B, 2023. [Online].
Available: https://www.ti.com/lit/ds/symlink/iwr6843aop.pdf

[16] Qorvo Inc., DWM3000 Ultra-Wideband Module Data Sheet, Rev. B, May 2021. [Online]. Available: https://www.qorvo.com/products/p/DWM3000

[17] Melexis N.V., MLX90640 Infrared Array (32×24) Datasheet, Rev. 14, Mar. 2024. [Online]. Available: https://www.melexis.com/-/media/files/documents/datasheets/mlx90640-datasheet-melexis.pdf

[18] Espressif Systems, ESP32-WROOM-32 Datasheet, Rev. 3.9, Feb. 2024. [Online].
Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

[19] Espressif Systems, ESP-IDF Programming Guide (Firmware & FreeRTOS API), 2024. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/

[20] NXP Semiconductors, UM10204 – I²C-Bus Specification and User Manual, Rev. 7, Oct. 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[21] RAiO Technology Inc., RA8875 TFT LCD Controller Datasheet, Jan. 10, 2012.

[22] EastRising Technology Co., Ltd., ER-TFTM043A1-7S TFT LCD Module Datasheet, Apr. 2, 2021. [Online].
Available: https://www.buydisplay.com/download/manual/ER-TFTM043A1-7S_Datasheet.pdf

[23] Bosch Sensortec, BMI270 IMU Datasheet, BST-BMI270-DS000-07, Mar. 13, 2023.

[24] Bosch Sensortec, BMM150 Magnetometer Datasheet, 2023.

[25] Bosch Sensortec, BMP390 – Digital Pressure Sensor Datasheet, Rev. 1.7, Mar. 2021.

[26] u-blox AG, ZED-F9R High Precision Sensor Fusion GNSS Receiver (ZED-F9R-03B) Data Sheet, Doc. UBX-22024085, Rev. R04, May 3, 2024.

[27] u-blox AG, NEO-M9V Multi-Mode Dead Reckoning Module Data Sheet (NEO-M9V-20B), Doc. UBX-21029781, Rev. R07, Jun. 2, 2025.

[28] M5Stack, Module GNSS – SKU M135 (Product Wiki and Datasheet), Aug. 2025.

[29] Sercel, SM-24 Geophone Element Brochure, Carquefou, France, 2012. [Online].
Available: https://cdn.sparkfun.com/datasheets/Sensors/Accelerometers/SM-24%20Brochure.pdf

[30] Seis Tech Co., Ltd., Data Sheet of SM-24 10 Hz Equivalent Geophone, Shenzhen, China, 2024. [Online]. Available: https://www.seis-tech.com/wp-content/uploads/2024/08/data-sheet-of-sm24-10hz-equivalent-geophone.pdf

[31] D. H. Titterton and J. L. Weston, Strapdown Inertial Navigation Technology, 2nd ed., Stevenage, U.K.: IET, 2004.

[32] P. D. Groves, Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, 2nd ed., Boston, MA: Artech House, 2013.

[33] O. J. Woodman, “An Introduction to Inertial Navigation,” Univ. of Cambridge, Tech. Rep. UCAM-CL-TR-696, Aug. 2007.

[34] E. Foxlin, “Inertial Head-Tracker Sensor Fusion by a Complementary Separate-Bias Kalman Filter,” Proc. IEEE Virtual Reality Symp., 1996, pp. 185–194.

[35] R. Zetik, J. Sachs, and R. Thomä, “UWB Localization—Advantages and Challenges,” Proc. IEEE Int. Conf. Ultra-Wideband (ICUWB), Sep. 2004, pp. 771–778.

[36] F. Montorsi, M. Shubair, and R. Fantacci, “Cooperative Localization Techniques for UWB Wireless Sensor Networks,” IEEE Commun. Mag., vol. 53, no. 6, pp. 124–131, Jun. 2015.

[37] S. Lazaro, D. Girbau, and R. Villarino, “Analysis of Vital Signs Monitoring Using an IR-UWB Radar,” Progress In Electromagnetics Research, vol. 100, pp. 265–284, 2010.

[38] J. Sachs, R. Zetik, and R. Thomä, “Through-Wall Imaging with UWB Radar,” Proc. IEEE European Radar Conf., Oct. 2005, pp. 561–564.

[39] International Telecommunication Union (ITU-R), Recommendation M.1477 – Technical and Performance Requirements for GNSS Receivers, Geneva, Switzerland, 2017.

[40] U.S. Department of Defense, IS-GPS-200H: Navstar GPS Space Segment/Navigation User Interfaces, El Segundo, CA, USA, 2019.

[41] Elecfreaks, "Ultrasonic Ranging Module HC-SR04," SparkFun Electronics, Boulder, CO, USA, Datasheet, 2014. [Online]. Available:
https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

[42] Adafruit Industries, "HC-SR04 Ultrasonic Sonar Distance Sensor," Adafruit Learning System, New York, NY, USA, 2023. [Online]. Available: https://learn.adafruit.com/ultrasonic-sonar-distance-sensors

[43] List of Unclassified Manufacturers, "HC-SR501 PIR Motion Detector," AllDatasheet, Datasheet, 2013. [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/1131987/ETC2/HC-SR501.html

[44] HandsOnTec, "HC-SR501 Passive Infrared (PIR) Motion Sensor," HandsOnTec Dataspecs, 2022. [Online]. Available: http://www.handsontec.com/dataspecs/SR501%20Motion%20Sensor.pdf

[45] STMicroelectronics, "VL53L0X Time-of-Flight Ranging Sensor," STMicroelectronics, Geneva, Switzerland, Datasheet, Rev. 1, 2015. [Online]. Available: https://www.st.com/resource/en/datasheet/vl53l0x.pdf

[46] Pololu Corporation, "VL53L0X Time-of-Flight Distance Sensor Carrier," Pololu Product Documentation, Las Vegas, NV, USA, 2023. [Online]. Available: https://www.pololu.com/product/2490

[47] Texas Instruments, "IWR6843AOP Single-Chip 60- to 64-GHz mmWave Sensor With Antennas-on-Package (AOP)," Texas Instruments, Dallas, TX, USA, Datasheet, Rev. B, 2022. [Online]. Available: https://www.ti.com/lit/ds/symlink/iwr6843aop.pdf

[48] Texas Instruments, "60-GHz mmWave Sensor Evaluation Module (EVM) User's Guide," Texas Instruments, Dallas, TX, USA, User's Guide, Rev. E, 2023. [Online]. Available: https://www.ti.com/lit/ug/swru546e/swru546e.pdf

[49] Federal Communications Commission, "Radio Frequency Safety," FCC, Washington, DC, USA, OET Bulletin 65, Ed. 97-01, 1997. [Online]. Available: https://www.fcc.gov/general/radio-frequency-safety-0

[50] Federal Communications Commission, "47 CFR § 1.1310 - Radiofrequency Radiation Exposure Limits," Electronic Code of Federal Regulations, Washington, DC, USA, 2023. [Online]. Available: https://www.law.cornell.edu/cfr/text/47/1.1310

Appendix B - Datasheets

1. M135 INS Module - https://docs.m5stack.com/en/module/GNSS%20Module
2. Radar Module - https://www.ti.com/lit/ds/swrs237c/swrs237c.pdf?ts=1763590269728&ref_

url=https%253A%252F%252Fwww.ti.com%252Ftool%252FIWR6843AOPEVM
3. DWM3000 UWB Module - https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.qorvo.com/products/d/da008334&ved=2ahUKEwjm46_-pIGRAxUIRTABHU1tKxsQFnoECAwQAQ&usg=AOvVaw25qNFnxwPRz0Dh--_C0vZx
4. USS HC-SR04 - https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf&ved=2ahUKEwianpn6pYGRAxXKRzABHe5RD94QFnoECBoQAQ&usg=AOvVaw1iQg0OJ6MFfs9MrkZYGAB4
5. IR MLX90640 - https://www.melexis.com/en/documents/documentation/datasheets/datasheet-mlx90640
6. ESP32-WROOM-32E - https://documentation.espressif.com/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf
7. 3.2-inch LCD screen - https://www.lcdwiki.com/3.2inch_SPI_Module_ILI9341_SKU:MSP3218
8. APEM Buttons - https://www.atakelstore.com/image/data/pdf/IP_mom.pdf
9. Buzzer - https://www.sameskydevices.com/product/resource/cmt-1203-smt-tr.pdf?srsltid=AfmBOooz9YujWpwn6Drf7Ph2u0ygIY_gb6lLc2IjaKb78_paCcyETgw9
10. 3.3-Volt Regulator – https://www.pololu.com/product/2122
11. 5-Volt Regulator - https://www.pololu.com/product/2123
12. USS HC-SR04 - https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

Appendix C - Software Code

1. Code for IR sensor, LCD, and USS Prototype.

```
#include <SPI.h>
#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#include <XPT2046_Touchscreen.h>
#include <Wire.h>
#include "MLX90640_API.h"
#include "MLX90640_I2C_Driver.h"

#define TFT_CS    14
#define TFT_DC    27
#define TFT_RST   33
#define TOUCH_CS   5
#define TOUCH_IRQ  4

Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC, TFT_RST);
```

```
XPT2046_Touchscreen ts(TOUCH_CS, TOUCH_IRQ);

#define MLX90640_ADDR 0x33
#define EMISSIVITY 0.95
#define TA_SHIFT 8
paramsMLX90640 mlx90640;
float pixels[834];
const int SCALE = 13;

#define TRIG_PIN 12
#define ECHO_PIN 13

enum Mode { MENU, THERMAL, PROXIMITY };
Mode currentMode = MENU;

// COLOR PALETTE (28–42°C)
uint16_t humanColor(float temp) {
  temp = constrain(temp, 28.0f, 42.0f);
  float v = (temp - 28.0f) / 14.0f;  // 0.0 → 1.0

  uint8_t r = 0, g = 0, b = 0;
  if (v <= 0.35f) {                 // 28–33°C → cold blue → cyan
    b = 255;
    g = v * (255.0f / 0.35f);
  }
  else if (v <= 0.65f) {            // 33–37°C → cyan → yellow
    b = 255 - ((v - 0.35f) / 0.3f) * 255;
    g = 255;
    r = (v - 0.35f) / 0.3f * 200;
  }
  else {                            // 37–42°C → yellow → BRIGHT RED
    r = 255;
    g = (1.0f - (v - 0.65f) / 0.35f) * 255;
    b = 0;
  }
  return tft.color565(r, g, b);
}

void setup() {
  Serial.begin(115200);
  tft.begin();
  tft.setRotation(1);
  tft.fillScreen(ILI9341_BLACK);

  ts.begin();
  ts.setRotation(1);

  Wire.begin();
  Wire.setClock(1000000);
  uint16_t ee[832];
  MLX90640_DumpEE(MLX90640_ADDR, ee);
  MLX90640_ExtractParameters(ee, &mlx90640);
  MLX90640_SetRefreshRate(MLX90640_ADDR, 0x05);
```

```
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);

  drawMenu();
}

void loop() {
  if (ts.tirqTouched() && ts.touched()) {
    handleTouch();
    delay(300);
    while (ts.touched()) delay(10);
  }

  if (currentMode == THERMAL) runThermal();
  if (currentMode == PROXIMITY) runProximity();
}

void handleTouch() {
  if (currentMode != MENU) {
    currentMode = MENU;
    drawMenu();
    return;
  }
  TS_Point p = ts.getPoint();
  int x = map(p.x, 200, 3800, 0, 320);
  int y = map(p.y, 200, 3800, 0, 240);

  if (x > 20 && x < 150 && y > 40 && y < 200) {
    currentMode = PROXIMITY;
    tft.fillScreen(ILI9341_BLACK);
  }
  else if (x > 170 && x < 300 && y > 40 && y < 200) {
    currentMode = THERMAL;
    tft.fillScreen(ILI9341_BLACK);
  }
}

void drawMenu() {
  tft.fillScreen(ILI9341_BLACK);
  tft.fillRoundRect(20, 40, 130, 160, 20, ILI9341_NAVY);
  tft.drawRoundRect(20, 40, 130, 160, 20, ILI9341_CYAN);
  tft.setTextColor(ILI9341_WHITE); tft.setTextSize(2);
  tft.setCursor(40, 100); tft.println("THERMAL");

  tft.fillRoundRect(170, 40, 130, 160, 20, ILI9341_MAROON);
  tft.drawRoundRect(170, 40, 130, 160, 20, ILI9341_RED);
  tft.setCursor(185, 100); tft.println("PROXIMITY");

  tft.setTextSize(1); tft.setTextColor(ILI9341_CYAN);
  tft.setCursor(80, 220); tft.println("Tap tile to select");
}
```

```
void runThermal() {
  static unsigned long last = 0;
  if (millis() - last < 125) return;  // 8 FPS
  last = millis();

  uint16_t frame[834];
  for (uint8_t i = 0; i < 2; i++) {
    if (MLX90640_GetFrameData(MLX90640_ADDR, frame) < 0) continue;
    float Ta = MLX90640_GetTa(frame, &mlx90640);
    MLX90640_CalculateTo(frame, &mlx90640, EMISSIVITY, Ta - TA_SHIFT, pixels);
  }

 // SMOOTH PIXELS
  for (int y = 0; y < 24; y++) {
    for (int x = 0; x < 32; x++) {
      float temp = pixels[y * 32 + x];
      uint16_t col = humanColor(temp);
      int px = x * SCALE + 2;
      int py = y * SCALE + 8;
      tft.fillRoundRect(px, py, SCALE-2, SCALE-2, 4, col);
    }
  }

  tft.setTextSize(1); tft.setTextColor(ILI9341_WHITE);
  tft.setCursor(5, 5);
  tft.print("HUMAN MODE 28-42C   Tap=Menu");
}

 void runProximity() {
  digitalWrite(TRIG_PIN, LOW); delayMicroseconds(2);
  digitalWrite(TRIG_PIN, HIGH); delayMicroseconds(10);
  digitalWrite(TRIG_PIN, LOW);
  long dur = pulseIn(ECHO_PIN, HIGH, 30000);
  float dist = dur * 0.0343 / 2;

  tft.fillScreen(ILI9341_BLACK);
  tft.setTextSize(3); tft.setTextColor(ILI9341_CYAN);
  tft.setCursor(50, 60); tft.println("PROXIMITY");

  if (dist > 0 && dist < 25) {
    tft.fillRect(0, 130, 320, 80, ILI9341_RED);
    tft.setTextColor(ILI9341_WHITE);
    tft.setCursor(45, 155); tft.println("TOO CLOSE!");
  } else {
    tft.setTextColor(ILI9341_GREEN);
    tft.setCursor(30, 150);
    tft.print("Dist: "); tft.print(dist > 0 ? dist : 999, 1); tft.println("cm");
  }
  tft.setTextColor(ILI9341_WHITE); tft.setTextSize(1);
  tft.setCursor(2, 2); tft.print("Tap to go back");
  delay(200);
 }
```

2. Final Integrated Code

```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#include "MLX90640_API.h"
#include "MLX90640_I2C_Driver.h"
#include "thermal_human_detector.h"
#include "scan_mode.h"
#include <math.h>
#include <MPU9250.h>
#include <Adafruit_BMP280.h>
#include <MadgwickAHRS.h>
#include <TinyGPSPlus.h>
#include <string.h>
#include <pgmspace.h>
#include <esp32-hal-cpu.h>
#include "Radar2BLEClient.h"   // NimBLE client — fuses Radar2 vitals with TI radar
#define MAIN_U1
#include "uwb.h"
#include "dw3000.h"
#include "saver.h"
#include <NimBLEDevice.h>
#define CONFIG_BT_NIMBLE_MAX_CONNECTIONS 4
#include <vector>

// ---- BLE (NimBLE-Arduino) ----
// dw3000_port.h defines BIT_0..BIT_31, READ, WRITE, etc. which clash with
// NimBLE / ESP-IDF symbols.  Undef them all before including NimBLE.
#undef BIT_0
#undef BIT_1
#undef BIT_2
#undef BIT_3
#undef BIT_4
#undef BIT_5
#undef BIT_6
#undef BIT_7
#undef BIT_8
#undef BIT_9
#undef BIT_10
#undef BIT_11
#undef BIT_12
#undef BIT_13
#undef BIT_14
#undef BIT_15
#undef BIT_16
```

```
#undef BIT_17
#undef BIT_18
#undef BIT_19
#undef BIT_20
#undef BIT_21
#undef BIT_22
#undef BIT_23
#undef BIT_24
#undef BIT_25
#undef BIT_26
#undef BIT_27
#undef BIT_28
#undef BIT_29
#undef BIT_30
#undef BIT_31
// dw3000_port.h also defines READ/WRITE as bare 0x00/0x01 constants which
// corrupt the NIMBLE_PROPERTY enum inside NimBLELocalValueAttribute.h.
#undef READ
#undef WRITE

// ---- audio ----
#include "AudioFileSourcePROGMEM.h"
#include "AudioGeneratorWAV.h"
#include "AudioOutputI2S.h"
#include "take1.h"
#include "hd.h"
#include "buttonclick.h"
#include "tooclosef.h"
#include "welcome.h"
#define I2S_BCLK  26
#define I2S_LRCLK 27
#define I2S_DIN   14


// ----------------- UI / MODE DECLARATIONS -----------------
// Top-level app modes, navigation states, and shared drawing helpers.
struct TileCfg;
enum Mode : uint8_t { MENU, THERMAL_LOADING, SCAN_LOADING,
THERMAL, PROXIMITY, INS_MODE, SCAN_MODE };
enum NavButton { NAV_NONE, NAV_LEFT, NAV_RIGHT, NAV_UP,
NAV_DOWN, NAV_CENTER };
enum ToolbarButton { TOOLBAR_CLOSE = 0, TOOLBAR_START = 1,
TOOLBAR_MARK = 2 };
enum AudioClip    { CLIP_NONE, CLIP_SCAN, CLIP_HUMAN, CLIP_CLICK,
CLIP_TOOCLOSE, CLIP_WELCOME };
```

```
enum AudioState   { AUDIO_IDLE, AUDIO_STARTING, AUDIO_PLAYING,
AUDIO_FINISHING, AUDIO_DONE };
enum ScanReviewScreen : uint8_t { SCAN_REVIEW_SUMMARY = 0,
SCAN_REVIEW_THERMAL = 1, SCAN_REVIEW_NAV = 2 };
extern Mode currentMode;

bool hitTile(int x, int y, const TileCfg& t);
void drawTile(const TileCfg& t);
void drawBleTile();
void drawMenu();
void toggleBleConnection();
void bleSendLog(const char* msg);
void bleSendThermalFrame();
void bleSendRadarData();
void bleSendInsData();
void drawTopBar(const char *title, bool showStart, bool showMark);
void deselectSharedSpiDevices();
void deselectSharedSpiDevicesLocked();
void setupUWB();
void lcdBeginAccess();
void lcdEndAccess();
void uwbBeginAccess();
void uwbEndAccess();
bool uwbTryBeginAccess();
NavButton readNavButton();
void handleNavButton(NavButton btn);
void setStartOrigin();
void markVictim();
void updateThermalDetection();
void runScan();
float computeCompassHeadingDeg();
void drawLoadingScreen();
void updateLoadingBar(float progress);
void stopAudioNow();
void resetAudioState();
void startClip(AudioClip clip);
bool isAudioMutedForCurrentScreen();
bool alertAllowedForCurrentScreen();
void triggerAlert();
void resetAlert();
void serviceAudio();
void serviceAlert();
void serviceTooClose();
void serviceLED();
void playWelcomeIfIdle();
bool tooCloseWarningActive();
```

```
void catDigitalWriteP1(uint8_t bit, bool level);
void resetProximityReading();
static void triggerLED();
static inline float emaUpdate(float prev, float sample, float alpha);

extern bool radarEnabled;
extern bool proxTooCloseLatched;
extern bool tooCloseAlertPlaying;

// Screen Saver
static const uint32_t IDLE_TIMEOUT_MS = 10000;
uint32_t lastInputMs = 0;
bool screensaverOn = false;

// LCD
static const int TFT_CS = 25;
static const int TFT_DC = 32;
static const int TFT_SCLK = 18;
static const int TFT_MOSI = 23;
static const int TFT_MISO = 19;
static const int TOUCH_IRQ = 34;
Adafruit_ILI9341 tft(TFT_CS, TFT_DC, -1);
static const SPISettings LCD_SPI_SETTINGS(40000000, MSBFIRST,
SPI_MODE0);
static const SPISettings UWB_SPI_SETTINGS(2000000, MSBFIRST,
SPI_MODE0);
static const int TOUCH_X_MIN = 200;
static const int TOUCH_X_MAX = 3800;
static const int TOUCH_Y_MIN = 200;
static const int TOUCH_Y_MAX = 3800;

// CAT9555 definitions
static const uint8_t CAT9555_ADDR = 0x20;
static const uint8_t REG_IN0 = 0x00;
static const uint8_t REG_IN1 = 0x01;
static const uint8_t REG_OUT0 = 0x02;
static const uint8_t REG_OUT1 = 0x03;
static const uint8_t REG_CFG0 = 0x06;
static const uint8_t REG_CFG1 = 0x07;
static const uint8_t TFT_RST_BIT = 3;
static const uint8_t TOUCH_CS_BIT = 7;
static const uint8_t USS_ECHO_BIT = 0;
static const uint8_t USS_TRIG_BIT = 1;
static const uint8_t UWB_RST_BIT = 2;
static const uint8_t BTN_LEFT_BIT = 6;
static const uint8_t BTN_UP_BIT = 5;
```

```
static const uint8_t BTN_RIGHT_BIT = 0;
static const uint8_t BTN_DOWN_BIT = 3;
static const uint8_t BTN_CENTER_BIT = 2;
static const uint8_t LED1_BIT = 7;


//Thermal
static uint32_t ledOnUntilMs = 0;      // millis() timestamp when LED should turn
back off; 0=LED idle
bool prevThermalValid = false;

// ----------------- SCAN / FUSION -----------------
// The scan screen runs the staged fusion flow defined in scan_mode.h
ScanMode::ScanRuntimeState scanRt;
ScanMode::ScanConfig scanCfg = ScanMode::defaultConfig();
bool scanScreenInit = false;
char prevScanLine[10][48];
uint16_t prevScanProgressFill = 0;
ScanReviewScreen scanReviewScreen = SCAN_REVIEW_SUMMARY;
bool scanReviewSequenceActive = false;
bool scanReviewCompletionLatched = false;
bool thermalHumanDetected = false;

// ----------------- AUDIO SYSTEM -----------------
// Audio is played from PROGMEM WAV arrays using ESP8266Audio.
// startClip() selects the WAV source and starts the decoder.
// serviceAudio() must be called often from loop() so the decoder can keep
// the I2S buffer full. If serviceAudio() is starved by TFT/radar/USS work, audio
may stutter, repeat, or sound pitch-shifted.
// Long UI updates are paused during important clips such as scan loading,
human detection, and the too-close warning.
static const int FADE_SAMPLES = 512;

class VoiceTunedI2S : public AudioOutputI2S {
public:
  volatile float gainRamp = 0.0f;

  bool ConsumeSample(int16_t sample[2]) override {
    int32_t mono = ((int32_t)sample[0] + (int32_t)sample[1]) / 2;
    mono = (int32_t)((float)mono * gainRamp);

    if (mono > 32767) mono = 32767;
    if (mono < -32768) mono = -32768;

    sample[0] = (int16_t)mono;
    sample[1] = (int16_t)mono;
```

```
    return AudioOutputI2S::ConsumeSample(sample);
  }
};

// I2S output wrapper used to force stereo WAV output into centered mono and
apply a lightweight gain ramp. gainRamp changes volume only.

VoiceTunedI2S          *audioOut  = nullptr;
AudioGeneratorWAV      *wav       = nullptr;
AudioFileSourcePROGMEM *audioFile = nullptr;

AudioState audioState   = AUDIO_IDLE;
AudioClip  currentClip  = CLIP_NONE;
int        fadeSample   = 0;
uint32_t   audioStartMs = 0;

// Alert state -- plays hd.h ONCE per THERMAL session
bool     alertArmed     = false;
bool     alertTriggered = false;
bool     alertPlayed    = false;

void loadClip(AudioClip clip) {
  if (audioFile) { delete audioFile; audioFile = nullptr; }

  if (clip == CLIP_SCAN) {
    audioFile = new AudioFileSourcePROGMEM(take1_wav, take1_wav_len);
  } else if (clip == CLIP_HUMAN) {
    audioFile = new AudioFileSourcePROGMEM(detected_wav,
detected_wav_len);
  } else if (clip == CLIP_CLICK) {
    audioFile = new AudioFileSourcePROGMEM(buttonclick_wav,
buttonclick_wav_len);
  } else if (clip == CLIP_TOOCLOSE) {
    audioFile = new AudioFileSourcePROGMEM(tooclosef_wav,
tooclosef_wav_len);
  } else if (clip == CLIP_WELCOME) {
    audioFile = new AudioFileSourcePROGMEM(welcome_wav,
welcome_wav_len);
  }
  currentClip = clip;
}


volatile bool audioBusy = false;
// Starts a new PROGMEM WAV clip
```

```
// This owns the AudioFileSourcePROGMEM lifetime and selects the clip data
based on the AudioClip enum. The actual playback is advanced by
serviceAudio().
void startClip(AudioClip clip) {
    if (audioBusy) return;
    audioBusy = true;

    if (!audioOut || !wav) {
        audioBusy = false;
        return;
    }

    loadClip(clip);

    if (!audioFile) {
        audioBusy = false;
        return;
    }

    wav->begin(audioFile, audioOut);

    if (clip == CLIP_CLICK) {
    audioOut->gainRamp   = 0.0f;
    fadeSample           = FADE_SAMPLES;

  } else if (clip == CLIP_HUMAN) {
    audioOut->gainRamp   = 0.0f;
    fadeSample           = 0;

 } else if (clip == CLIP_TOOCLOSE) {
    audioOut->gainRamp   = 1.0f;
    fadeSample           = FADE_SAMPLES;


  } else {
    audioOut->gainRamp   = 1.0f;
    fadeSample           = FADE_SAMPLES;

  }
  audioState = AUDIO_STARTING;
  audioStartMs = millis();
  audioBusy = false;
}

void stopAudioNow() {
  if (audioBusy) return;
```

```
  audioBusy = true;
  if (wav && wav->isRunning()) wav->stop();
  if (audioOut) audioOut->gainRamp = 0.0f;
  fadeSample  = 0;
  audioState  = AUDIO_IDLE;
  audioBusy = false;
}

void resetAudioState() {
  stopAudioNow();
  currentClip = CLIP_NONE;
}

void playWelcomeIfIdle() {
    if (audioState == AUDIO_IDLE || audioState == AUDIO_DONE) {
        startClip(CLIP_WELCOME);
    }
}

bool tooCloseWarningActive() {
    return currentMode == PROXIMITY &&
           currentClip == CLIP_TOOCLOSE &&
           (audioState == AUDIO_STARTING ||
            audioState == AUDIO_PLAYING);
}

// Advances the WAV decoder state machine.
// This function should be called frequently from loop(). It feeds decoded samples
to I2S, handles clip startup, optional gain fade, and marks clips
// AUDIO_DONE when playback finishes.
void serviceAudio() {
  if (!audioOut || !wav) return;

  switch (audioState) {
    case AUDIO_IDLE:
      break;

case AUDIO_STARTING:
  wav->loop();

  if (wav->isRunning()) {
    audioState = AUDIO_PLAYING;

    if (currentClip == CLIP_TOOCLOSE) {
      fadeSample = FADE_SAMPLES;
      audioOut->gainRamp = 1.0f;
```

```
  } else {
    fadeSample = FADE_SAMPLES;
    audioOut->gainRamp = 1.0f;
  }
} else if (millis() - audioStartMs > 200) {
  audioState = AUDIO_DONE;
}
break;

  case AUDIO_PLAYING:
    if (currentClip != CLIP_TOOCLOSE && fadeSample < FADE_SAMPLES) {
      fadeSample++;
      audioOut->gainRamp = (float)fadeSample / (float)FADE_SAMPLES;
    } else if (currentClip == CLIP_TOOCLOSE) {
      audioOut->gainRamp = 1.0f;
    }
  if (wav->isRunning()) {
    for (int i = 0; i < 4 && wav->isRunning(); i++) {
      wav->loop();
    }

    } else {
      if (currentClip == CLIP_TOOCLOSE) {
        audioOut->gainRamp = 0.0f;
        audioState = AUDIO_DONE;
      } else {
        audioState = AUDIO_FINISHING;
        fadeSample = FADE_SAMPLES;
      }
    }
    break;

  case AUDIO_FINISHING:
    if (currentClip == CLIP_TOOCLOSE) {
      audioOut->gainRamp = 0.0f;
      audioState = AUDIO_DONE;
      break;
    }

    if (fadeSample > 0) {
      fadeSample--;
      audioOut->gainRamp = (float)fadeSample / (float)FADE_SAMPLES;

      if (wav->isRunning()) {
        for (int i = 0; i < 4 && wav->isRunning(); i++) {
          wav->loop();
```

```
        }
       }
      } else {
       audioOut->gainRamp = 0.0f;
       audioState = AUDIO_DONE;
      }
      break;

    case AUDIO_DONE:
      break;
  }
}


// Alert: arms on THERMAL entry, fires once on first human detection, disarms.
void resetAlert() {
  alertArmed     = true;
  alertTriggered = false;
  alertPlayed    = false;
}

bool alertAllowedForCurrentScreen() {
  if (currentMode == THERMAL) return true;

  if (currentMode == SCAN_MODE) {
    // The scan thermal screen is shown from SCAN_IR through SCAN_FUSION
and
    // during the post-scan thermal review. Allow the HUMAN clip to fire only
    // while the thermal map is actually on-screen. We intentionally do NOT
    // include a bare SCAN_DONE here: scanRt.state sits at SCAN_DONE for the
    // entire review sequence (THERMAL -> NAV -> SUMMARY) and a bare
DONE
    // clause would let the clip re-fire and paint the green banner over the
    // NAV or final summary screen the moment the operator walked back in
    // front of the MLX. The (reviewSeqActive && reviewScreen==THERMAL) leg
    // covers the legitimate post-DONE thermal review window — the only time
    // the review screen actually shows the thermal map.
    return scanRt.state == ScanMode::SCAN_IR       ||
           scanRt.state == ScanMode::SCAN_POSITION ||
           scanRt.state == ScanMode::SCAN_FUSION   ||
           (scanReviewSequenceActive && scanReviewScreen ==
SCAN_REVIEW_THERMAL);
  }

  return false;
}
```

```
void triggerAlert() {
  if (!alertAllowedForCurrentScreen()) return;
  if (!alertArmed || alertTriggered || alertPlayed) return;
  alertArmed     = false;
  alertTriggered = true;
  startClip(CLIP_HUMAN);
  triggerLED();
}

void serviceAlert() {
  if (!alertTriggered) return;

  if (audioState == AUDIO_DONE && currentClip == CLIP_HUMAN) {
    alertPlayed    = true;
    alertTriggered = false;
    resetAudioState();

    // In SCAN_MODE the HUMAN alert is one-shot per scan: once it has played,
    // the operator is expected to review the thermal map and press NEXT —
    // re-firing the clip (e.g. because the subject briefly walked out of the
    // MLX frame and back in) would be disruptive. A fresh scan's START
    // handler calls resetAlert() which re-arms for the next cycle.
    if (currentMode == SCAN_MODE) return;

    // Stand-alone THERMAL screen: if the person disappeared while the alert
    // was playing, allow the next confirmed detection to trigger the
    // banner/audio again. This is the original live-detector behaviour.
    if (!thermalHumanDetected) {
      alertArmed  = true;
      alertPlayed = false;
    }
  }
}



// USS too-close warning lifecycle.
// When proxTooCloseLatched becomes true in PROXIMITY mode (or during a
SCAN
// radar/IR/position/fusion phase — the operator can still walk into USS range
// on the scan screen), this state machine interrupts lower-priority audio,
// starts CLIP_TOOCLOSE, and keeps the LED on.
// While the warning clip is active, runProximity()/runScan() show the TOO
// CLOSE banner and skip the heavy radar/TFT rendering path.
// Once the clip finishes, the current USS flag is checked. If still too close,
```

```
// the clip restarts; if clear, the warning releases and the screen resumes.

void serviceTooClose() {
   // Mirror the PROXIMITY behaviour onto the scan screen: the same USS
   // too-close flag is already produced by updateProximityReading() during
   // runScan(), so we just need to let the warning lifecycle fire here too.
   if (currentMode != PROXIMITY && currentMode != SCAN_MODE) return;
   if (!proxTooCloseLatched) return;
   // Re-fire whenever not already playing so the clip loops while still too close.
   if (!tooCloseAlertPlaying && (audioState == AUDIO_IDLE || audioState ==
AUDIO_DONE)) {
      stopAudioNow();
      lcdBeginAccess();
      tft.fillScreen(ILI9341_RED);
      tft.setTextSize(3);
      tft.setTextColor(ILI9341_WHITE, ILI9341_RED);
      tft.setCursor(46, 105);
      tft.print("TOO CLOSE!");
      lcdEndAccess();
      startClip(CLIP_TOOCLOSE);
      tooCloseAlertPlaying = true;
   }
}
// LED timeout service.
// triggerLED() sets ledOnUntilMs in the future. serviceLED() keeps LED1 on
// until that timestamp expires, then turns it off. Some safety states, like
// TOO CLOSE, directly hold the LED on by clearing ledOnUntilMs and writing
ON.

void serviceLED() {
   if (ledOnUntilMs > 0) {
      if (millis() < ledOnUntilMs) {
         catDigitalWriteP1(LED1_BIT, true);   // active high = ON
      } else {
         catDigitalWriteP1(LED1_BIT, false);  // active high = OFF
         ledOnUntilMs = 0;                    // return to idle
      }
   }
}
static void triggerLED() {
   ledOnUntilMs = millis() + 5000;
}

// ----------------- THERMAL -----------------
// MLX90640 frame buffers and detector state used by the thermal screen.
#define MLX90640_ADDR 0x33
```

```
#define EMISSIVITY 0.95f
#define TA_SHIFT 8
paramsMLX90640 mlx90640;
float mlxRaw[768];
float mlxFilt[768];
static constexpr int THERM_RENDER_W = 64;
static constexpr int THERM_RENDER_H = 48;
static constexpr int THERM_PIXEL_W = 4;
static constexpr int THERM_PIXEL_H = 4;
static constexpr uint8_t MLX_ACTIVE_REFRESH = 0x02;
static constexpr uint8_t MLX_IDLE_REFRESH   = 0x00;
float *upscaled = nullptr;
bool mlxReady = false;
bool mlxInitDone = false;
uint8_t thermalWarmupFrames = 0;
bool thermalImagePrimed = false;
float thermalAmbientC = NAN;
float thermalSceneAvgC = NAN;
float thermalMaxC = NAN;
int thermalHotPixelCount = 0;
bool thermalFrameValid = false;
ThermalHuman::ThermalState thermalDetectorState;
ThermalHuman::ThermalConfig thermalDetectorCfg =
ThermalHuman::hybridDefaults();
ThermalHuman::ThermalFeatures thermalDetectorFeatures;
ThermalHuman::ThermalScores thermalDetectorScores;

// ----------------- INS / IMU / BARO / GNSS -----------------
// Navigation state, drift-controlled display position, and user markers.
static constexpr int SDA_PIN = 21;
static constexpr int SCL_PIN = 22;
static constexpr int GPS_RX_PIN = 16;
static constexpr int GPS_TX_PIN = 17;
static constexpr float G = 9.80665f;
static constexpr float SEA_LEVEL_HPA = 1019.0f;
MPU9250 mpu;
Adafruit_BMP280 bmp;
Madgwick filter;
HardwareSerial RadarDataSerial(1);
TinyGPSPlus gps;
bool gnssActive = false;
uint32_t gnssByteCount = 0;
uint32_t gnssLastByteMs = 0;
char gnssPreview[33] = "";
uint8_t gnssPreviewLen = 0;
bool imuReady = false;
```

```
bool bmpReady = false;
uint32_t insPrevUs = 0;
uint8_t stillSampleCount = 0;
uint32_t insResetHoldUntilMs = 0;
uint32_t lastInsTrackMs = 0;
float accBiasX = 0, accBiasY = 0, accBiasZ = 0;
float gyroZBias = 0;
float yawDeg = 0;
bool yawInit = false;
float compassHeadingDeg = NAN;
bool compassValid = false;
float localX_m = 0.0f;
float localY_m = 0.0f;
float displayX_m = 0.0f;
float displayY_m = 0.0f;
float vE = 0.0f, vN = 0.0f;
float aELpf = 0.0f, aNLpf = 0.0f;
float rollDegNowAbs = 0.0f, pitchDegNowAbs = 0.0f;
float rollDegDisp = 0.0f, pitchDegDisp = 0.0f, yawDegDisp = 0.0f;
float rollZero = 0.0f, pitchZero = 0.0f, yawZero = 0.0f;
static constexpr float AXY_LPF = 0.20f;
static constexpr float AXY_DEADBAND = 0.08f;
static constexpr float VEL_DAMP = 0.995f;
static constexpr float VEL_MAX = 3.0f;
static constexpr float STILL_ACC_BAND = 0.12f;
static constexpr float STILL_GYRO_DPS = 1.8f;
static constexpr uint8_t STILL_CONFIRM_SAMPLES = 8;
static constexpr float DISPLAY_MOVE_THRESH_M = 0.20f;
static constexpr float BMP_TEMP_DISPLAY_OFFSET_C = 5.0f;
bool localOriginSet = false;
double lat0 = 0.0, lon0 = 0.0;
double lastLat = 0.0, lastLon = 0.0;
bool gpsValidNow = false;
float temperatureC = NAN;
float insAmbientDisplayC = NAN;
float pressureHpa = NAN;
float altitudeM = NAN;
float altitudeZeroM = NAN;
bool startSet = false;
float startX = 0, startY = 0;
static const int TRACK_MAX = 220;
struct Pt { float x, y; };
Pt track[TRACK_MAX];
int trackCount = 0;
int trackHead = 0;
static const int VICTIM_MAX = 40;
```

```
Pt victims[VICTIM_MAX];
int victimCount = 0;
static const int VICTIM_LOG_MAX = 4;
struct VictimSnapshot {
    float e_m;
    float n_m;
    float yaw_deg;
    float alt_m;
    float temp_c;
    float pressure_hpa;
    bool gpsFix;
    double lat;
    double lon;
    uint32_t tMs;
};
VictimSnapshot victimLog[VICTIM_LOG_MAX];
int victimLogCount = 0;
int victimLogHead = 0;
static const int MAP_W = 41;
static const int MAP_H = 14;
static const float MAP_SPAN_M = 200.0f;
char prevInsGrid[MAP_H][MAP_W + 1];
char prevInsLine[13][80];
bool insCacheInit = false;

// ----------------- RADAR -----------------
// Radar parser state and filtered outputs for the proximity screen.
static constexpr int RADAR_RX_PIN = 16;
static constexpr int RADAR_TX_PIN = 17;
static constexpr int RADAR_DATA_RX_PIN = 33;
static constexpr int RADAR_DATA_TX_PIN = -1;
static constexpr int RADAR_NRST_PIN = 15;
static constexpr uint32_t RADAR_CLI_BAUD = 115200;
static constexpr uint32_t RADAR_DATA_BAUD = 921600;
static constexpr float RADAR_STOP_THRESH_CM = 20.0f;
static constexpr float RADAR_BIN_TO_M = 0.05f;
static constexpr float USS_MAX_STEP_CM = 18.0f;
static constexpr float USS_CAL_GAIN = 1.52f;
static constexpr float USS_DISPLAY_STEP_CM = 1.0f;
static constexpr uint32_t RADAR_TLV_LOG_PERIOD_MS = 1000;
static constexpr uint32_t RADAR_TLV_VITALSIGNS = 1040;
static constexpr uint32_t RADAR_TLV_VITALSIGNS_LEN = 136;
static constexpr uint32_t RADAR_TLV_RANGE_PROFILE = 2;
static constexpr uint32_t RADAR_TLV_RANGE_DOPPLER_HEATMAP = 5;
static constexpr uint32_t RADAR_TLV_DETECTED_POINTS_COMPRESSED =
301;
```

```
static constexpr uint32_t RADAR_TLV_RANGE_AZIMUTH_HEATMAP_MAJOR
= 304;
static constexpr uint32_t RADAR_TLV_RANGE_AZIMUTH_HEATMAP_MINOR
= 305;
static constexpr uint32_t RADAR_TLV_TARGET_LIST = 1010;
static constexpr uint32_t RADAR_TLV_PRESENCE = 1021;
static constexpr uint32_t RADAR_PHASE_MS = 30000;
static constexpr uint32_t RADAR_LAST_DISTANCE_WINDOW_MS = 10000;
static constexpr uint32_t RADAR_PRESENCE_TIMEOUT_MS = 10000;
static constexpr uint32_t RADAR_CMD_DELAY_MS = 25;
static constexpr uint32_t RADAR_CMD_DELAY_FINAL_MS = 80;
static constexpr uint32_t RADAR_BOOT_SETTLE_MS = 250;
static constexpr uint32_t RADAR_PROMPT_TIMEOUT_MS = 2500;
static constexpr uint32_t RADAR_PROX_BOOT_WAIT_MS = 1200;
static constexpr uint32_t RADAR_PROX_PROMPT_TIMEOUT_MS = 1200;
static constexpr uint32_t RADAR_PROX_CMD_DELAY_MS = 20;
static constexpr uint32_t RADAR_MAX_FRAME_LEN = 12000;
// These must match the live profileCfg the scan/proximity radar sends:
// "profileCfg 0 60.75 30.00 1.00 10.00 0 0 200.0 1 96 10785.00 2 1 36"
//                                  ^slope       ^N  ^digOutSR
// The previous values (65, 4000, 200) were leftovers from an older config
// and caused radarRangeBinToMeters() to under-report distance by ~1.8x,
// which is why the DIST card never lined up with a real bin even on the
// fallback path.
static constexpr float RADAR_PROFILE_SLOPE_MHZ_PER_US = 200.0f;
static constexpr float RADAR_ADC_SAMPLE_RATE_KSPS    = 10785.0f;
static constexpr float RADAR_ADC_SAMPLES             = 96.0f;
static const int RADAR_PROFILE_MAX_BINS = 128;
static const int RADAR_DISTANCE_SAMPLE_MAX = 512;

enum RadarScanState : uint8_t {
   RADAR_SCAN_IDLE,
   RADAR_SCAN_WAIT_P1,
   RADAR_SCAN_RUNNING_P1,
   RADAR_SCAN_WAIT_P2,
   RADAR_SCAN_RUNNING_P2,
   RADAR_SCAN_DONE
};

struct RadarTimedDistanceSample {
   float valueM;
   uint32_t tMs;
};
// shared port, radar config send at boot then port goes to INS
HardwareSerial RadarSerial(2);
HardwareSerial& GNSS = RadarSerial;
```

```
bool radarEnabled = false;
bool radarDataMode = false;
bool radarHuman = false;
bool radarTargetPresent = false;
bool radarVitalsReady = false;
bool proxRadarScopeBaseDrawn = false;
float radarBreathBpm = NAN;
float radarHeartBpm = NAN;
float radarDistM = NAN;
float radarBreathDisplayBpm = NAN;
float radarHeartDisplayBpm = NAN;
float radarDistDisplayM = NAN;
// Extra display-side smoothing for the radar (proximity) screen. The scan
// screen gets its steadiness from the 30 s phase-1 running average, but the
// radar screen has no phase — it's always "live". A simple EMA on top of the
// already-median-filtered radarHeartBpm / radarBreathBpm gives that screen a
// comparable, calm read without waiting 30 s.
// Time constants: tau_s / (tau_s + dt) at ~10 Hz and tau≈6 s → alpha ≈ 0.015
// (per sample) for HR, tau≈8 s for BR → alpha ≈ 0.012.
static constexpr float RADAR_DISPLAY_EMA_ALPHA_HR = 0.015f;
static constexpr float RADAR_DISPLAY_EMA_ALPHA_BR = 0.012f;
float radarHeartBpmDisplay  = NAN;
float radarBreathBpmDisplay = NAN;

// ---- Vital-sign median filter ------------------------------------------
// The TI radar's per-frame HR/BR estimate is very noisy (±10-20 bpm swings are
// common between consecutive TLVs). Radar firmware is fixed, but we can still
// clean up the downstream values by passing each valid sample through a small
// median-of-N window before it reaches the scan phase-1 average / the card.
// A 9-sample window at ~10 Hz gives ~1 s of smoothing, which is enough to kill
// isolated spikes without adding meaningful lag.
//
// Design notes (the tightness of an earlier version caused the phase-1 average
// to latch high because early high samples became the reference median and
// every later real sample then looked like an outlier):
//   1. No outlier rejection is applied until the window is completely full —
//      during warm-up we accept every sample so the median can discover the
//      real distribution instead of anchoring on whatever came in first.
//   2. The jump limits are generous (±25 bpm for HR, ±12 bpm for BR) and only
//      rejection OVER that limit. Real heart rates can drift a lot between
//      frames when the radar re-locks, so this stays loose.
//   3. If we reject too many samples in a row (baseline probably drifted or
//      the subject changed) we flush the window and let it warm back up on
//      the new data instead of staying stuck.
static constexpr uint8_t RADAR_VITAL_MEDIAN_LEN = 9;
```

```
static constexpr uint8_t RADAR_VITAL_STUCK_LIMIT = 6;
float radarHrMedianBuf[RADAR_VITAL_MEDIAN_LEN];
float radarBrMedianBuf[RADAR_VITAL_MEDIAN_LEN];
uint8_t radarHrMedianCount = 0;
uint8_t radarBrMedianCount = 0;
uint8_t radarHrMedianHead  = 0;
uint8_t radarBrMedianHead  = 0;
uint8_t radarHrRejectStreak = 0;
uint8_t radarBrRejectStreak = 0;

static inline void radarResetVitalMedians() {
    radarHrMedianCount = 0;
    radarBrMedianCount = 0;
    radarHrMedianHead  = 0;
    radarBrMedianHead  = 0;
    radarHrRejectStreak = 0;
    radarBrRejectStreak = 0;
}

static inline float radarMedianOf(const float* buf, uint8_t count) {
    if (count == 0) return NAN;
    float tmp[RADAR_VITAL_MEDIAN_LEN];
    for (uint8_t i = 0; i < count; i++) tmp[i] = buf[i];
    // Small N, insertion sort is fine and avoids bringing in <algorithm>.
    for (uint8_t i = 1; i < count; i++) {
        float k = tmp[i];
        int8_t j = (int8_t)i - 1;
        while (j >= 0 && tmp[j] > k) { tmp[j + 1] = tmp[j]; j--; }
        tmp[j + 1] = k;
    }
    return tmp[count / 2];
}

// Returns the updated HR median after accepting `bpm`. During warm-up every
// sample is accepted; once the window is full, samples further than ±25 bpm
// from the current median are rejected, and if rejections stack up we flush
// the window so a real baseline shift can take over.
static inline float radarPushHrSample(float bpm) {
    bool warm = radarHrMedianCount >= RADAR_VITAL_MEDIAN_LEN;
    if (warm) {
        float prev = radarMedianOf(radarHrMedianBuf, radarHrMedianCount);
        if (isfinite(prev) && fabsf(bpm - prev) > 25.0f) {
            radarHrRejectStreak++;
            if (radarHrRejectStreak < RADAR_VITAL_STUCK_LIMIT) return prev;
            // Too many rejections in a row — baseline probably moved, reset.
            radarHrMedianCount = 0;
```

```
            radarHrMedianHead  = 0;
            radarHrRejectStream = 0;
        }
    }
    radarHrRejectStreak = 0;
    radarHrMedianBuf[radarHrMedianHead] = bpm;
    radarHrMedianHead = (radarHrMedianHead + 1) %
RADAR_VITAL_MEDIAN_LEN;
    if (radarHrMedianCount < RADAR_VITAL_MEDIAN_LEN)
radarHrMedianCount++;
    return radarMedianOf(radarHrMedianBuf, radarHrMedianCount);
}

// Same pattern as HR. Breathing is slower and naturally smoother so the
// window can be tighter without starving the filter of samples.
static inline float radarPushBrSample(float bpm) {
    bool warm = radarBrMedianCount >= RADAR_VITAL_MEDIAN_LEN;
    if (warm) {
        float prev = radarMedianOf(radarBrMedianBuf, radarBrMedianCount);
        if (isfinite(prev) && fabsf(bpm - prev) > 12.0f) {
            radarBrRejectStreak++;
            if (radarBrRejectStreak < RADAR_VITAL_STUCK_LIMIT) return prev;
            radarBrMedianCount = 0;
            radarBrMedianHead  = 0;
            radarBrRejectStreak = 0;
        }
    }
    radarBrRejectStreak = 0;
    radarBrMedianBuf[radarBrMedianHead] = bpm;
    radarBrMedianHead = (radarBrMedianHead + 1) %
RADAR_VITAL_MEDIAN_LEN;
    if (radarBrMedianCount < RADAR_VITAL_MEDIAN_LEN)
radarBrMedianCount++;
    return radarMedianOf(radarBrMedianBuf, radarBrMedianCount);
}
float radarDistCandidateM = NAN;
float radarBreathCandidateBpm = NAN;
float radarHeartCandidateBpm = NAN;
float radarPhase1HrSum = 0.0f;
float radarPhase1BrSum = 0.0f;
float radarPhase1HrAvg = NAN;
float radarPhase1BrAvg = NAN;
float radarPhase2DistAvg = NAN;
uint8_t radarDistCandidateCount = 0;
uint8_t radarBreathCandidateCount = 0;
uint8_t radarHeartCandidateCount = 0;
```

```
uint16_t radarPhase1SampleCount = 0;
uint16_t radarPhase2SampleCount = 0;
uint16_t radarLastRangeBin = 0;
uint32_t radarLastPacketMs = 0;
uint32_t radarLastPresenceMs = 0;
uint32_t radarLastVitals1040Ms = 0;
uint32_t radarPhaseStartMs = 0;
RadarScanState radarScanState = RADAR_SCAN_IDLE;
RadarTimedDistanceSample
radarPhase2Samples[RADAR_DISTANCE_SAMPLE_MAX];
char radarLastTlvSummary[48] = "TLV:none";
char radarDistanceHint[48] = "D?:--";
char radarScanStatus[48] = "Press START";
uint16_t radarRangeProfile[RADAR_PROFILE_MAX_BINS];
int radarRangeProfileCount = 0;
bool radarRangeProfileValid = false;
uint32_t radarRangeProfileMs = 0;
int radarPeakBin = -1;
float radarPeakScore = 0.0f;
uint8_t radarPeakCandidateCount = 0;
int radarPeakCandidateBin = -1;
uint32_t radarPeakRenderKey = 0;

// ------------------------UWB Defs----------------------------
// volatile ensures the compiler doesn't cache these in registers since
// they can be updated from both the main loop and interrupt context
volatile float uwbLatestDistanceM = NAN;      // most recent valid range
measurement in meters
volatile uint32_t uwbLatestUpdateMs = 0;       // millis() timestamp of last valid
range
volatile bool uwbLatestValid = false;          // true only when a fresh range is
available
volatile float uwbLastGoodDistanceM = NAN;     // last valid range ever — never
cleared on disable
float fusedDistanceM = NAN;                    // weighted blend of UWB and INS
distance estimates
bool uwbInitDone = false;                      // true after setupUWB() has been called
at least once
bool uwbInitOk = false;                        // true only if setupUWB() found and
configured the DW3000
TaskHandle_t uwbTaskHandle = nullptr;          // reserved for future FreeRTOS
task — not currently used
bool uwbTaskStarted = false;                   // reserved — not currently used
SemaphoreHandle_t spiBusMutex = nullptr;       // FreeRTOS mutex guarding the
shared SPI bus
```

```
bool uwbActive = false;                    // true when ranging should be attempted
each poll cycle
bool uwbStartPending = false;              // true when UWB init was requested
but not yet completed
bool uwbSpiPortReady = false;              // true after spiBegin/spiSelect have
been called once
bool uwbForcePollPending = false;          // set true to bypass the poll interval
timer on next loop
bool insSensorsActive = false;             // true when IMU/baro/GNSS are
powered and running
bool thermalActive = false;                // true when MLX90640 is in active refresh
mode
char uwbStatus[24] = "UWB:off";            // human-readable status string shown
on INS HUD

// ---------------RADAR ----------------------------
struct RadarTlvStat {
   uint32_t type;
   uint32_t count;
   uint32_t lastLen;
};
static const int RADAR_TLV_STAT_MAX = 8;
RadarTlvStat radarTlvStats[RADAR_TLV_STAT_MAX];
int radarTlvStatCount = 0;
uint32_t radarLastTlvPrintMs = 0;
uint32_t radarLastTlvDumpMs = 0;
uint32_t radarLastDumpedType = 0;
uint32_t radarLastDumpedLen = 0;

#define RADAR_BUF_SIZE 16384
uint8_t *radarBuf = nullptr;
int radarBufLen = 0;
const uint8_t RADAR_MAGIC[8] = {0x02,0x01,0x04,0x03,0x06,0x05,0x08,0x07};

// ----------------RADAR CONFIG--------------------

// Phase 1: optimised for heart-rate and breathing-rate acquisition.
const char* radarCfgPhase1[] = {
   "sensorStop", "flushCfg", "pmicCfg 1 1", "dfeDataOutputMode 1",
   "channelCfg 15 7 0", "adcCfg 2 1", "adcbufCfg -1 0 1 1 1", "lowPower 0 0",
   "profileCfg 0 60.75 30.00 1.00 10.00 0 0 200.0 1 96 10785.00 2 1 36",
   "chirpCfg 0 0 0 0 0 0 0 5", "chirpCfg 1 1 0 0 0 0 0 2", "chirpCfg 2 2 0 0 0 0 0 5",
   "frameCfg 0 2 48 0 90.00 1 0",
   "dynamicRACfarCfg -1 4 4 2 2 8 12 4 12 5.00 8.00 0.40 1 1",
   "staticRACfarCfg -1 6 2 2 2 8 8 6 4 8.00 15.00 0.30 0 0",
   "dynamicRangeAngleCfg -1 0.75 0.0010 1 0",
```

```
    "dynamic2DAngleCfg -1 1.5 0.0300 1 0 1 0.30 0.85 8.00",
    "staticRangeAngleCfg -1 0 8 8", "fineMotionCfg -1 1", "bpmCfg -1 1 0 1",
    "antGeometry0 -1 -1 0 0 -3 -3 -2 -2 -1 -1 0 0",
    "antGeometry1 -1 0 -1 0 -3 -2 -3 -2 -3 -2 -3 -2",
    "antPhaseRot 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1", "fovCfg -1 70.0 20.0",
    "compRangeBiasAndRxChanPhase 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1
0 1 0 -1 0",
    "staticBoundaryBox -3 3 0.3 5.5 0 3", "boundaryBox -4 4 0.5 6 0 3",
    "sensorPosition 2 0 15", "gatingParam 3 2 2 2 10",
    "stateParam 3 3 12 215 50 500", "allocationParam 5 50 0.1 20 0.5 20",
    "trackingCfg 1 2 800 1 46 96 90", "presenceBoundaryBox -3 3 0.3 5.5 0 3",
    "maxAcceleration 0.1 0.1 0.1", "vitalsign 1 300", "VSRangeIdxCfg 1 10",
    "sensorStart", nullptr
};

// Phase 2: optimised for distance accuracy.
const char* radarCfgPhase2[] = {
    "sensorStop", "flushCfg", "pmicCfg 1 1", "dfeDataOutputMode 1",
    "channelCfg 15 7 0", "adcCfg 2 1", "adcbufCfg -1 0 1 1 1", "lowPower 0 0",
    "profileCfg 0 60.75 30.00 1.00 10.00 0 0 200.0 1 96 10785.00 2 1 36",
    "chirpCfg 0 0 0 0 0 0 0 5", "chirpCfg 1 1 0 0 0 0 0 2", "chirpCfg 2 2 0 0 0 0 0 5",
    "frameCfg 0 2 48 0 90.00 1 0",
    "dynamicRACfarCfg -1 4 4 2 2 8 12 4 12 5.00 8.00 0.40 1 1",
    "staticRACfarCfg -1 6 2 2 2 8 8 6 4 8.00 15.00 0.30 0 0",
    "dynamicRangeAngleCfg -1 0.75 0.0010 1 0",
    "dynamic2DAngleCfg -1 1.5 0.0300 1 0 1 0.30 0.85 8.00",
    "staticRangeAngleCfg -1 0 8 8", "fineMotionCfg -1 1", "bpmCfg -1 1 0 2",
    "antGeometry0 -1 -1 0 0 -3 -3 -2 -2 -1 -1 0 0",
    "antGeometry1 -1 0 -1 0 -3 -2 -3 -2 -3 -2 -3 -2",
    "antPhaseRot 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1", "fovCfg -1 35.0 20.0",
    "compRangeBiasAndRxChanPhase 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1
0 1 0 -1 0",
    "staticBoundaryBox -3.0 3.0 0.5 8.0 0 3.0", "boundaryBox -3.0 3.0 0.5 8.0 0
3.0",
    "presenceBoundaryBox -3.0 3.0 0.5 8.0 0 3.0",
    "sensorPosition 0 0 0", "gatingParam 4 3 3 3 12",
    "stateParam 3 3 12 215 50 6000", "allocationParam 10 80 0.1 20 0.5 20",
    "trackingCfg 1 2 800 1 46 96 90", "maxAcceleration 0.2 0.2 0.2",
    "vitalsign 15 300", "VSRangeIdxCfg 0 21",
    "sensorStart", nullptr
};

static inline uint32_t rdU32(const uint8_t* p) {
    uint32_t v;
    memcpy(&v, p, 4);
    return v;
```

```
}
static inline uint16_t rdU16(const uint8_t* p) {
    return (uint16_t)p[0] | ((uint16_t)p[1] << 8);
}
static inline float rdF32(const uint8_t* p) {
    float f;
    memcpy(&f, p, sizeof(float));
    return f;
}
static inline float radarRangeBinToMeters(uint16_t rangeBin) {
    const float sampleRateHz = RADAR_ADC_SAMPLE_RATE_KSPS * 1000.0f;
    const float slopeHzPerSec = RADAR_PROFILE_SLOPE_MHZ_PER_US *
1.0e12f;
    const float rangeRes = (299792458.0f * sampleRateHz) / (2.0f *
slopeHzPerSec * RADAR_ADC_SAMPLES);
    return rangeBin * rangeRes;
}
static inline float radarCurrentDistanceM() {
    if (isfinite(radarDistDisplayM)) return radarDistDisplayM;
    if (isfinite(radarDistM)) return radarDistM;
    if (radarPeakBin >= 0) return
radarRangeBinToMeters((uint16_t)radarPeakBin);
    return NAN;
}
static inline void radarUpdateDistance(float sampleM, bool authoritative) {
    if (!isfinite(sampleM) || sampleM <= 0.05f || sampleM >= 30.0f) return;
    const float immediateBandM = authoritative ? 0.18f : 0.12f;
    const float candidateBandM = authoritative ? 0.22f : 0.16f;
    const uint8_t acceptCount = authoritative ? 2 : 3;
    if (!isfinite(radarDistDisplayM)) {
        radarDistDisplayM = sampleM;
        radarDistM = sampleM;
        radarDistCandidateM = NAN;
        radarDistCandidateCount = 0;
        return;
    }
    if (fabsf(sampleM - radarDistDisplayM) <= immediateBandM) {
        radarDistDisplayM = emaUpdate(radarDistDisplayM, sampleM,
authoritative ? 0.18f : 0.10f);
        radarDistM = radarDistDisplayM;
        radarDistCandidateM = NAN;
        radarDistCandidateCount = 0;
        return;
    }
    if (isfinite(radarDistCandidateM) && fabsf(sampleM - radarDistCandidateM) <=
candidateBandM) {
```

```
        radarDistCandidateCount++;
        if (radarDistCandidateCount >= acceptCount) {
            radarDistDisplayM = emaUpdate(radarDistDisplayM, sampleM,
authoritative ? 0.12f : 0.08f);
            radarDistM = radarDistDisplayM;
            radarDistCandidateM = NAN;
            radarDistCandidateCount = 0;
        }
    } else {
        radarDistCandidateM = sampleM;
        radarDistCandidateCount = 1;
    }
}
Mode currentMode = MENU;
Mode lastMode = MENU;
void applyPeripheralMode(Mode mode);
void ensureUwbActive(bool enable);
void ensureRadarActive(bool enable);
void ensureThermalActive(bool enable);
void ensureInsSensorsActive(bool enable);
bool ensureThermalBuffers();
void releaseThermalBuffers();
bool ensureRadarBuffer();
void releaseRadarBuffer();
void radarRecordTlv(uint32_t type, uint32_t tlvLen);
void radarPrintTlvStats();
void radarDumpTlvSample(uint32_t type, const uint8_t* payload, uint32_t tlvLen);
void radarStart();
void radarStop();
void radarStartProximity();
void radarProximityArmIdle();
const char* radarScanStateName();
uint16_t radarScanProgressPermille();
bool touchDown = false;
bool proxScreenInit = false;
bool proxRadarRestartPrompt = false;
bool proxRadarHeldInReset = true;
bool insScreenInit = false;
bool thermalHudInit = false;
bool topBarDirty = true;
bool menuDrawn = false;
char topBarTitleCache[16] = "";
bool topBarShowStartCache = false;
bool topBarShowMarkCache = false;
ToolbarButton topBarSelectionCache = TOOLBAR_CLOSE;
char topBarStartLabelCache[8] = "";
```

```
int presenceScore = 0;
int thermalConfidencePct = 0;
float proxFiltered = -1.0f;
bool proxTooCloseLatched = false;
bool tooCloseAlertPlaying = false;
uint32_t proxLastSampleMs = 0;
uint8_t proxInvalidCount = 0;
uint8_t proxCloseCount = 0;
int menuSelection = 0;
ToolbarButton toolbarSelection = TOOLBAR_CLOSE;
bool scanThermalReviewInit = false;
bool scanNavReviewInit = false;
char prevScanPhaseLabel[40] = "";
char prevScanPhaseTimer[12] = "";
bool humanIndicatorShown = false;
uint16_t *prevThermal = nullptr;

static inline bool scanUiShouldIgnoreAudioBlock() {
   return currentMode == SCAN_MODE &&
         (scanReviewSequenceActive || scanRt.state ==
ScanMode::SCAN_DONE);
}
static inline bool scanModeShowsMarkButton() {
   return currentMode == SCAN_MODE &&
         scanReviewSequenceActive &&
         scanReviewScreen == SCAN_REVIEW_NAV;
}
static inline float scanDisplayConfidence() {
   // Once fuseResults() has run (SCAN_FUSION / SCAN_DONE), show the
latched
   // overall score so the decision text and the percentage agree.
   //
   // Before fusion runs we blend the live radar confidence with the live
   // thermal confidence using the same IR-heavy weights the fusion engine
   // uses (wr=0.35, wi=0.55). This keeps the "Final" line and the
   // Human/Possible/No-human decision updating in real time through the
   // RADAR and IR phases, and avoids the earlier problem where we were
   // mirroring just the thermal value and mislabelling it as "Final".
   if (scanRt.fusion.completed) {
      return scanRt.fusion.overallConfidence;
   }
   const float radar = scanRt.radar.radarConfidence;
   const float ir    = (float)thermalConfidencePct / 100.0f;
   constexpr float wr = 0.35f;
   constexpr float wi = 0.55f;
   float f = (radar * wr + ir * wi) / (wr + wi);
```

```
    if (f < 0.0f) f = 0.0f;
    if (f > 1.0f) f = 1.0f;
    return f;
}

// -------------------MENU---------------------
struct TileCfg {
    int x;
    int y;
    int w;
    int h;
    uint16_t fill;
    uint16_t border;
    const char* label;
    int labelX;
    int labelY;
    bool selectable;
    Mode mode;
};
static const uint16_t TILE_ORANGE = 0xEB85;
TileCfg TILE_THERM = { 10, 36, 145, 72, TILE_ORANGE, ILI9341_WHITE,
"THERMAL", 37, 64, true, THERMAL };
TileCfg TILE_USS = {165, 36, 145, 72, TILE_ORANGE, ILI9341_WHITE,
"RADAR", 205, 64, true, PROXIMITY };
TileCfg TILE_INS = { 10, 118, 145, 72, TILE_ORANGE, ILI9341_WHITE, "INS",
64, 146, true, INS_MODE };
TileCfg TILE_FUS = {165, 118, 145, 72, TILE_ORANGE, ILI9341_WHITE,
"SCAN", 212, 146, true, SCAN_MODE };
static const int BTN_Y = 2;
static const int BTN_H = 20;
static const int BTN_START_X = 154, BTN_START_W = 70;
static const int BTN_MARK_X = 228, BTN_MARK_W = 46;
static const int BTN_CLOSE_X = 278, BTN_CLOSE_W = 40;
static const int INS_INFO_Y = 158;
static const int INS_INFO_DY = 8;
static const int RADAR_INFO_Y = 194;
static const int RADAR_INFO_DY = 10;
static const int BLE_TILE_X = 40;
static const int BLE_TILE_Y = 198;
static const int BLE_TILE_W = 240;
static const int BLE_TILE_H = 36;

// BLE connection state
enum BleState : uint8_t { BLE_DISCONNECTED, BLE_ADVERTISING,
BLE_CONNECTED };
BleState bleState = BLE_DISCONNECTED;
```

```
bool bleNeedsRedraw = false;
// Characteristic UUIDs:
#define BLE_THERMAL_CHAR_UUID "7a3b4c5d-6e7f-8a9b-0c1d-
2e3f4a5b6c7d"
// Chunk framing:
static const int BLE_THERMAL_CHUNK_DATA = 490;

NimBLEServer*         bleServer         = nullptr;
NimBLECharacteristic* bleCharacteristic = nullptr;  // log channel
NimBLECharacteristic* bleCharThermal    = nullptr;  // thermal frame channel
std::vector<uint16_t> bleConnHandles; // active connection handle
bool                  bleSendConnectedLog = false; // true after connect until log is
sent
uint32_t              bleConnectedMs    = 0;       // millis() of last connection

static inline float clampf(float v, float lo, float hi) {
  return (v < lo) ? lo : (v > hi ? hi : v);
}
static inline float wrap360(float a) {
    while (a < 0.0f) a += 360.0f;
    while (a >= 360.0f) a -= 360.0f;
    return a;
}
static inline float emaUpdate(float prev, float sample, float alpha) {
    if (!isfinite(sample)) return prev;
    if (!isfinite(prev)) return sample;
    return prev + alpha * (sample - prev);
}
static void radarInitResetPin() {
    pinMode(RADAR_NRST_PIN, OUTPUT);
    digitalWrite(RADAR_NRST_PIN, HIGH);
}
static void radarPulseReset() {
    radarInitResetPin();
    digitalWrite(RADAR_NRST_PIN, LOW);
    delay(10);
    digitalWrite(RADAR_NRST_PIN, HIGH);
}
static inline bool vitalsConsistent(float sample, float baseline, float maxJump) {
    if (!isfinite(sample)) return false;
    if (!isfinite(baseline)) return true;
    return fabsf(sample - baseline) <= maxJump;
}
static float waveformPeakToPeak(const uint8_t* payload, int samples) {
    if (!payload || samples <= 0) return 0.0f;
    float lo = rdF32(payload + 0);
```

```
    float hi = lo;
    for (int i = 1; i < samples; i++) {
        float v = rdF32(payload + i * 4);
        if (v < lo) lo = v;
        if (v > hi) hi = v;
    }
    return hi - lo;
}
static float waveformMeanAbsDelta(const uint8_t* payload, int samples) {
    if (!payload || samples < 2) return 0.0f;
    float prev = rdF32(payload + 0);
    float sum = 0.0f;
    for (int i = 1; i < samples; i++) {
        float v = rdF32(payload + i * 4);
        sum += fabsf(v - prev);
        prev = v;
    }
    return sum / (samples - 1);
}

//-------------------INS/UWB SETTINGS----------------------
static constexpr float PITCH_GAIN = 1.11f;
static constexpr bool GNSS_DEBUG = false;
static constexpr bool UWB_DEBUG = true;
static constexpr uint32_t UWB_POLL_INTERVAL_MS = 10;
static constexpr uint32_t UWB_DISPLAY_HOLD_MS = 5000;
static inline bool allowSerialDebug() {
    return true;
}
static inline float correctPitchDegrees(float pitchDeg) {
    float corrected = pitchDeg * PITCH_GAIN;
    return clampf(corrected, -89.9f, 89.9f);
}
// GPs coordinates to meters
float latToMeters(double dLatDeg) { return (float)(dLatDeg * 110540.0); }
float lonToMeters(double dLonDeg, double latDeg) { return (float)(dLonDeg *
111320.0 * cos(latDeg * DEG_TO_RAD)); }
float mToFeet(float m) { return m * 3.280839895f; } // convert meters to feet
float cToF(float c) { return c * 9.0f / 5.0f + 32.0f; } // ºC to ºF
float hPaToInHg(float h) { return h * 0.0295299831f; }
void formatFeetInches(float meters, char *out, size_t outLen) {
    if (!isfinite(meters)) {
        snprintf(out, outLen, "---");
        return;
    }
    bool neg = meters < 0.0f;
```

```
  float totalIn = fabsf(meters) * 39.3700787f;
  int ft = (int)(totalIn / 12.0f);
  int in = (int)roundf(totalIn - (ft * 12.0f));
  if (in == 12) { ft += 1; in = 0; }
  if (neg) snprintf(out, outLen, "-%d'%d\"", ft, in);
  else snprintf(out, outLen, "%d'%d\"", ft, in);
}
void pumpGNSS() {
 while (GNSS.available()) {
  int ch = GNSS.read();
  if (ch < 0) break;
  gps.encode((char)ch);
  gnssByteCount++;
  gnssLastByteMs = millis();
  if (gnssPreviewLen < sizeof(gnssPreview) - 1) {
   char c = (char)ch;
   if (c == '\r' || c == '\n') {
    if (gnssPreviewLen > 0) {
     gnssPreview[gnssPreviewLen] = '\0';
     gnssPreviewLen = sizeof(gnssPreview) - 1;
    }
   } else if (c >= 32 && c <= 126) {
    gnssPreview[gnssPreviewLen++] = c;
    gnssPreview[gnssPreviewLen] = '\0';
   }
  }
 }
}
uint8_t catReadReg(uint8_t reg) {
  Wire.beginTransmission(CAT9555_ADDR);
  Wire.write(reg);
  Wire.endTransmission(false);
  Wire.requestFrom((int)CAT9555_ADDR, 1);
  return Wire.available() ? Wire.read() : 0xFF;
}
void catWriteReg(uint8_t reg, uint8_t val) {
  Wire.beginTransmission(CAT9555_ADDR);
  Wire.write(reg);
  Wire.write(val);
  Wire.endTransmission();
}
void catSetPinModeP0(uint8_t bit, bool output) {
  uint8_t cfg = catReadReg(REG_CFG0);
  if (output) cfg &= ~(1u << bit);
  else cfg |= (1u << bit);
  catWriteReg(REG_CFG0, cfg);
```

```
}
void catSetPinModeP1(uint8_t bit, bool output) {
    uint8_t cfg = catReadReg(REG_CFG1);
    if (output) cfg &= ~(1u << bit);
    else cfg |= (1u << bit);
    catWriteReg(REG_CFG1, cfg);
}
void catDigitalWriteP0(uint8_t bit, bool level) {
    uint8_t out = catReadReg(REG_OUT0);
    if (level) out |= (1u << bit);
    else out &= ~(1u << bit);
    catWriteReg(REG_OUT0, out);
}
void catDigitalWriteP1(uint8_t bit, bool level) {
    uint8_t out = catReadReg(REG_OUT1);
    if (level) out |= (1u << bit);
    else out &= ~(1u << bit);
    catWriteReg(REG_OUT1, out);
}
bool catDigitalReadP0(uint8_t bit) {
    return (catReadReg(REG_IN0) & (1u << bit)) != 0;
}
bool catDigitalReadP1(uint8_t bit) {
    return (catReadReg(REG_IN1) & (1u << bit)) != 0;
}

// thermal buffers
bool ensureThermalBuffers() {
    const size_t upscaledCount = THERM_RENDER_W * THERM_RENDER_H;
    if (!upscaled) {
        upscaled = (float*)malloc(sizeof(float) * upscaledCount);
        if (!upscaled) return false;
    }
    if (!prevThermal) {
        prevThermal = (uint16_t*)malloc(sizeof(uint16_t) * upscaledCount);
        if (!prevThermal) {
            free(upscaled);
            upscaled = nullptr;
            return false;
        }
    }
    return true;
}
void releaseThermalBuffers() {
    if (upscaled) {
        free(upscaled);
```

```
        upscaled = nullptr;
    }
    if (prevThermal) {
        free(prevThermal);
        prevThermal = nullptr;
    }
    prevThermalValid = false;
}

//radar buffers
bool ensureRadarBuffer() {
    if (radarBuf) return true;
    radarBuf = (uint8_t*)malloc(RADAR_BUF_SIZE);
    return radarBuf != nullptr;
}
void releaseRadarBuffer() {
    if (radarBuf) {
        free(radarBuf);
        radarBuf = nullptr;
    }
    radarBufLen = 0;
}

// button reading
NavButton readNavButton() {
    uint8_t in1 = catReadReg(REG_IN1);
    bool left = (in1 & (1u << BTN_LEFT_BIT)) == 0;
    bool up = (in1 & (1u << BTN_UP_BIT)) == 0;
    bool right = (in1 & (1u << BTN_RIGHT_BIT)) == 0;
    bool down = (in1 & (1u << BTN_DOWN_BIT)) == 0;
    bool center = (in1 & (1u << BTN_CENTER_BIT)) == 0;
    if (left) return NAV_LEFT;
    if (right) return NAV_RIGHT;
    if (up) return NAV_UP;
    if (down) return NAV_DOWN;
    if (center) return NAV_CENTER;
    return NAV_NONE;
}

// ----------------- SENSOR POWER / MODE LIFECYCLE -----------------
// Each screen enables only the peripherals it needs. These helpers centralize
// wake/sleep/reset and cached-state cleanup for cleaner mode transitions.
void ensureThermalActive(bool enable) {
  if (!mlxReady) return;
  if (enable) {
    if (!ensureThermalBuffers()) return;
```

```
  } else {
    releaseThermalBuffers();
    thermalHumanDetected = false;
    thermalFrameValid = false;
  }
    if (thermalActive == enable) return;
    MLX90640_SetRefreshRate(MLX90640_ADDR, enable ?
MLX_ACTIVE_REFRESH : MLX_IDLE_REFRESH);
    thermalActive = enable;
    if (enable) {
        mlxInitDone = false;
        thermalWarmupFrames = 0;
        thermalImagePrimed = false;
        ThermalHuman::initState(thermalDetectorState);
        memset(&thermalDetectorFeatures, 0, sizeof(thermalDetectorFeatures));
        memset(&thermalDetectorScores, 0, sizeof(thermalDetectorScores));
        thermalConfidencePct = 0;
        presenceScore = 0;
    }
}
void ensureInsSensorsActive(bool enable) {
    if (enable != gnssActive) {
        GNSS.end();
        delay(20);
        if (enable) {
            gps = TinyGPSPlus();
            lastLat = 0.0;
            lastLon = 0.0;
            gnssByteCount = 0;
            gnssLastByteMs = 0;
            gnssPreview[0] = '\0';
            gnssPreviewLen = 0;
            GNSS.begin(9600, SERIAL_8N1, GPS_RX_PIN, GPS_TX_PIN);
            delay(20);
            while (GNSS.available()) GNSS.read();
        }
        gnssActive = enable;
    }
    gpsValidNow = false;
    if (imuReady) mpu.sleep(!enable);
    if (bmpReady) {
        bmp.setSampling(
            enable ? Adafruit_BMP280::MODE_NORMAL :
Adafruit_BMP280::MODE_SLEEP,
            enable ? Adafruit_BMP280::SAMPLING_X2 :
Adafruit_BMP280::SAMPLING_NONE,
```

```
            enable ? Adafruit_BMP280::SAMPLING_X16 :
Adafruit_BMP280::SAMPLING_NONE,
            enable ? Adafruit_BMP280::FILTER_X16 :
Adafruit_BMP280::FILTER_OFF,
            Adafruit_BMP280::STANDBY_MS_500
        );
    }
    if (enable) {
        insPrevUs = micros();
    }
    insSensorsActive = enable;
}
void ensureUwbActive(bool enable) {
    catSetPinModeP0(UWB_RST_BIT, true);
    if (!enable) {
        catDigitalWriteP0(UWB_RST_BIT, false);
        uwbLatestValid = false;
        uwbLatestDistanceM = NAN;
        fusedDistanceM = NAN;
        uwbActive = false;
        uwbStartPending = false;
        uwbInitDone = false;
        uwbInitOk = false;
        strncpy(uwbStatus, "UWB:disabled", sizeof(uwbStatus) - 1);
        uwbStatus[sizeof(uwbStatus) - 1] = '\0';
        prevInsLine[7][0] = '\0';
        return;
    }

    catDigitalWriteP0(UWB_RST_BIT, true);
    uwbLatestValid = false;
    uwbLatestDistanceM = NAN;
    fusedDistanceM = NAN;
    uwbStartPending = false;
    uwbActive = true;
    counter = 0;
    distance = NAN;

    if (!uwbInitDone) {
        setupUWB();
        uwbInitDone = true;
        prevInsLine[7][0] = '\0';
    }

    if (!uwbInitOk) {
        uwbActive = false;
```

```
    }
}
void ensureRadarActive(bool enable) {
    if (enable) {
        if (radarEnabled && radarDataMode) return;
        if (radarEnabled && !radarDataMode) {
            RadarSerial.begin(RADAR_CLI_BAUD, SERIAL_8N1, RADAR_RX_PIN,
RADAR_TX_PIN);
            RadarDataSerial.begin(RADAR_DATA_BAUD, SERIAL_8N1,
RADAR_DATA_RX_PIN, RADAR_DATA_TX_PIN);
            radarDataMode = true;
            radarBufLen = 0;
        }
    } else {
        radarStop();
    }
}
void applyPeripheralMode(Mode mode) {
  static Mode appliedMode = MENU;
  bool wantFusionSense = (mode == THERMAL || mode ==
THERMAL_LOADING ||
                          mode == PROXIMITY || mode == SCAN_LOADING ||
                          mode == SCAN_MODE);
  ensureThermalActive(wantFusionSense);
  if (!wantFusionSense) ensureRadarActive(false);
  if (mode != INS_MODE && mode != SCAN_MODE && mode != PROXIMITY) {
    ensureInsSensorsActive(false);
    ensureUwbActive(false);
  }
  if (mode == INS_MODE || mode == SCAN_MODE || mode == PROXIMITY) {
    ensureInsSensorsActive(true);
  }
  if (mode == SCAN_MODE) {
    // Keep the radar transport online while the SCAN tile is open so
    // radarPoll() and the scan cards can see live data. The timed two-phase
    // radar scan itself still begins only when the user presses START, which
    // calls radarStart().
    ensureUwbActive(false);
    ensureRadarActive(true);
  } else if (wantFusionSense) {
    ensureRadarActive(true);
  }
  if (mode == INS_MODE) deselectSharedSpiDevices();
  appliedMode = mode;
}
int toolbarButtonCountForMode(Mode mode) {
```

```
            if (mode == INS_MODE) return 3;
            if (mode == PROXIMITY) return 2;
            if (mode == SCAN_MODE) return scanModeShowsMarkButton() ? 3 :
2;
            if (mode == MENU) return 0;
            return 1;
}
void normalizeToolbarSelection() {
   int count = toolbarButtonCountForMode(currentMode);
   if (count <= 0) {
      toolbarSelection = TOOLBAR_CLOSE;
      return;
   }
   if ((int)toolbarSelection >= count) toolbarSelection = TOOLBAR_CLOSE;
}
const char* currentStartButtonLabel() {
   if (currentMode == SCAN_MODE &&
      scanReviewSequenceActive &&
      (scanReviewScreen == SCAN_REVIEW_THERMAL || scanReviewScreen
== SCAN_REVIEW_NAV)) {
      return "NEXT";
   }
   return "START";
}
void moveMenuSelection(int newSelection) {
   newSelection = constrain(newSelection, 0, 4);
   if (newSelection == menuSelection) return;

   int oldSelection = menuSelection;
   menuSelection = newSelection;

   if (menuDrawn) {
      lcdBeginAccess();

      switch (oldSelection) {
         case 0: drawTile(TILE_THERM); break;
         case 1: drawTile(TILE_USS); break;
         case 2: drawTile(TILE_INS); break;
         case 3: drawTile(TILE_FUS); break;
         case 4: drawBleTile(); break;
      }

      switch (menuSelection) {
         case 0: drawTile(TILE_THERM); break;
         case 1: drawTile(TILE_USS); break;
         case 2: drawTile(TILE_INS); break;
```

```
            case 3: drawTile(TILE_FUS); break;
            case 4: drawBleTile(); break;
        }

        lcdEndAccess();
    } else {
        drawMenu();
    }

    // Play the click only after TFT writes are finished.
    if (audioState == AUDIO_IDLE || audioState == AUDIO_DONE) {
        startClip(CLIP_CLICK);
    }
}

// Performs the selected top-bar action for the active mode.
// CLOSE returns to menu, START begins calibration/scan/radar, and MARK logs
a victim.

void activateToolbarSelection() {
    if (currentMode == MENU) return;
    normalizeToolbarSelection();
    if (toolbarSelection == TOOLBAR_CLOSE) {
    toolbarSelection = TOOLBAR_CLOSE;
    currentMode = MENU;
    return;
}

    if (currentMode == INS_MODE && toolbarSelection == TOOLBAR_START) {
    lcdBeginAccess();
    tft.fillRoundRect(60, 100, 200, 40, 8, ILI9341_NAVY);
    tft.drawRoundRect(60, 100, 200, 40, 8, ILI9341_CYAN);
    tft.setTextSize(2);
    tft.setTextColor(ILI9341_CYAN, ILI9341_NAVY);
    tft.setCursor(75, 114);
    tft.print("Calibrating...");
    lcdEndAccess();
        setStartOrigin();
        insScreenInit = false;  // force full INS redraw to clear the overlay
        insCacheInit = false;
        Serial.printf("[START] uwbInitOk=%d uwbInitDone=%d\n", uwbInitOk,
uwbInitDone);
        if (!uwbInitOk) {
            uwbStartPending = true;
            uwbInitDone = false;
            strncpy(uwbStatus, "UWB:init...", sizeof(uwbStatus) - 1);
```

```
            uwbStatus[sizeof(uwbStatus) - 1] = '\0';
            prevInsLine[7][0] = '\0';
        }
        return;
    }
            if (currentMode == INS_MODE && toolbarSelection ==
TOOLBAR_MARK) {
                markVictim();
                return;
              }
              if (currentMode == SCAN_MODE && toolbarSelection ==
TOOLBAR_START) {
            if (scanReviewSequenceActive) {
                if (scanReviewScreen == SCAN_REVIEW_THERMAL) {
                    stopAudioNow();
                    resetAudioState();
                    scanReviewScreen = SCAN_REVIEW_NAV;
                    scanThermalReviewInit = false;
                    scanNavReviewInit = false;
                    insScreenInit = false;
                    insCacheInit = false;
                    toolbarSelection = TOOLBAR_START;
                    topBarDirty = true;
                    return;
                }
                if (scanReviewScreen == SCAN_REVIEW_NAV) {
                    stopAudioNow();
                    resetAudioState();
                    scanReviewScreen = SCAN_REVIEW_SUMMARY;
                    scanReviewSequenceActive = false;
                    scanNavReviewInit = false;
                    scanScreenInit = false;
                    toolbarSelection = TOOLBAR_START;
                    topBarDirty = true;
                    return;
                }
            }
                  setStartOrigin();
                  ScanMode::resetRuntime(scanRt);
                  ScanMode::beginScan(scanRt, millis());
            scanReviewSequenceActive = false;
            scanReviewScreen = SCAN_REVIEW_SUMMARY;
            scanReviewCompletionLatched = false;
            scanThermalReviewInit = false;
            scanNavReviewInit = false;
            // Force the scan cards base to repaint cleanly for the new run.
```

```
            // Without this, a restart from the SUMMARY screen keeps the old
            // cached layout and the HR/BR/DIST cells can show stale text
            // from the previous scan's final values until the first cell
            // update fires.
            scanScreenInit = false;
            thermalHumanDetected = false;
            thermalFrameValid = false;
            prevThermalValid = false;
            thermalHudInit = false;
            presenceScore = 0;
            thermalConfidencePct = 0;
            ThermalHuman::initState(thermalDetectorState);
            memset(&thermalDetectorFeatures, 0, sizeof(thermalDetectorFeatures));
            memset(&thermalDetectorScores, 0, sizeof(thermalDetectorScores));
            // Re-arm the human-detected alert so the SCAN_IR phase can fire it
            // exactly like the stand-alone THERMAL screen does on entry.
            resetAlert();
                radarStart();
                if (!uwbInitOk) {
                    uwbStartPending = true;
                    uwbInitDone = false;
                    strncpy(uwbStatus, "UWB:init...", sizeof(uwbStatus) - 1);
                    uwbStatus[sizeof(uwbStatus) - 1] = '\0';
                } else {
                    uwbActive = true;
                    uwbForcePollPending = true;
                }
                for (int i = 0; i < 10; i++) prevScanLine[i][0] = '\0';
            prevScanPhaseLabel[0] = '\0';
            prevScanPhaseTimer[0] = '\0';
                prevScanProgressFill = 0;
                return;
              }
        if (currentMode == SCAN_MODE && toolbarSelection ==
TOOLBAR_MARK) {
            markVictim();
            return;
        }
        if (currentMode == PROXIMITY && toolbarSelection == TOOLBAR_START)
{
            // Kick off the full two-phase sweep (30 s P1 + 30 s P2) so the
            // progress bar / phase labels on the radar screen actually move.
            // radarCompletePhase2() below keeps the radar alive in PROXIMITY
            // mode so live distance + vitals continue streaming after DONE.
            if (!ensureRadarBuffer()) return;
            proxScreenInit = false;
```

```
            proxRadarScopeBaseDrawn = false;
            radarStart();
            return;
        }
}
void handleNavButton(NavButton btn) {
    if (btn == NAV_NONE) return;
    if (currentMode == MENU) {
        switch (btn) {
            case NAV_LEFT:
                if (menuSelection == 1) moveMenuSelection(0);
                else if (menuSelection == 3) moveMenuSelection(2);
                break;
            case NAV_RIGHT:
                if (menuSelection == 0) moveMenuSelection(1);
                else if (menuSelection == 2) moveMenuSelection(3);
                break;
            case NAV_UP:
                if (menuSelection == 2) moveMenuSelection(0);
                else if (menuSelection == 3) moveMenuSelection(1);
                else if (menuSelection == 4) moveMenuSelection(2);
                break;
            case NAV_DOWN:
                if (menuSelection == 0) moveMenuSelection(2);
                else if (menuSelection == 1) moveMenuSelection(3);
                else if (menuSelection == 2) moveMenuSelection(4);
                else if (menuSelection == 3) moveMenuSelection(4);
                break;
            case NAV_CENTER:
                if (menuSelection == 0) currentMode = THERMAL_LOADING;
                else if (menuSelection == 1) currentMode = PROXIMITY;
                else if (menuSelection == 2) currentMode = INS_MODE;
                else if (menuSelection == 3) currentMode = SCAN_LOADING;
                else if (menuSelection == 4) toggleBleConnection();
                break;
            default:
                break;
        }
        return;
    }
    int count = toolbarButtonCountForMode(currentMode);
    if (count <= 0) return;
    if ((currentMode == THERMAL ||
         currentMode == THERMAL_LOADING) && btn == NAV_CENTER) {
        toolbarSelection = TOOLBAR_CLOSE;
        stopAudioNow(); resetAudioState();
```

```
      ensureThermalActive(false); thermalHumanDetected=false;
presenceScore=0;
      currentMode = MENU;
      return;
   }
    if (currentMode == SCAN_LOADING && btn == NAV_CENTER) {
      toolbarSelection = TOOLBAR_CLOSE;
      stopAudioNow(); resetAudioState();
      currentMode = MENU;
      return;
   }
   switch (btn) {
      case NAV_LEFT:
      case NAV_UP:
         toolbarSelection = (ToolbarButton)(((int)toolbarSelection + count - 1) %
count);
         topBarDirty = true;
         drawTopBar(
            (currentMode == THERMAL || currentMode ==
THERMAL_LOADING) ? "THERMAL" :
            currentMode == PROXIMITY ? "RADAR" :
            (currentMode == SCAN_MODE || currentMode == SCAN_LOADING) ?
"SCAN" : "INS MAP",
            currentMode == INS_MODE || currentMode == SCAN_MODE ||
currentMode == PROXIMITY,
            currentMode == INS_MODE || scanModeShowsMarkButton()
         );
         break;
      case NAV_RIGHT:
      case NAV_DOWN:
         toolbarSelection = (ToolbarButton)(((int)toolbarSelection + 1) % count);
         topBarDirty = true;
         drawTopBar(
            (currentMode == THERMAL || currentMode ==
THERMAL_LOADING) ? "THERMAL" :
            currentMode == PROXIMITY ? "RADAR" :
            (currentMode == SCAN_MODE || currentMode == SCAN_LOADING) ?
"SCAN" : "INS MAP",
            currentMode == INS_MODE || currentMode == SCAN_MODE ||
currentMode == PROXIMITY,
            currentMode == INS_MODE || scanModeShowsMarkButton()
         );
         break;
      case NAV_CENTER:
      topBarDirty = true;
         activateToolbarSelection();
```

```
            break;
        default:
            break;
    }
}

// ----------------- SHARED SPI BUS OWNERSHIP -----------------
// TFT, touch, and DW3000 share the same SPI bus. The mutex acts as the bus
// token, and each entry path forces all CS lines inactive before use.

// Acquires the shared SPI bus for TFT drawing.
// The TFT, touch controller, and DW3000 share SPI lines, so this forces all
// chip-selects inactive before any LCD transaction.

void lcdBeginAccess() {
    if (spiBusMutex) xSemaphoreTake(spiBusMutex, portMAX_DELAY);
    deselectSharedSpiDevicesLocked();
    delayMicroseconds(50);
}

// Releases the shared SPI bus after TFT drawing.
// All chip-selects are returned inactive before giving up the mutex.
void lcdEndAccess() {
    deselectSharedSpiDevicesLocked();
    delayMicroseconds(50);
    if (spiBusMutex) xSemaphoreGive(spiBusMutex);
}

void uwbBeginAccess() {
    if (spiBusMutex) xSemaphoreTake(spiBusMutex, portMAX_DELAY);
    deselectSharedSpiDevicesLocked();
    delayMicroseconds(50);
}

bool uwbTryBeginAccess() {
    if (spiBusMutex && xSemaphoreTake(spiBusMutex, 0) != pdTRUE) {
        return false;
    }
    deselectSharedSpiDevicesLocked();
    delayMicroseconds(50);
    return true;
}

void uwbEndAccess() {
    deselectSharedSpiDevicesLocked();
    delayMicroseconds(50);
```

```
    if (spiBusMutex) xSemaphoreGive(spiBusMutex);
}
// Hardware reset and initialize the DW3000 only when the INS screen actually
// requests UWB. The last UWB status string is updated for the INS HUD.
void setupUWB() {
    uwbInitOk = false;
    strncpy(uwbStatus, "UWB:resetting", sizeof(uwbStatus) - 1);
    uwbStatus[sizeof(uwbStatus) - 1] = '\0';

    Serial.println("[setupUWB] === STARTING BRING-UP ===");

    // 1) Hardware reset via CAT9555
    catSetPinModeP0(UWB_RST_BIT, true);
    Serial.println("[setupUWB] RST LOW (assert)");
    catDigitalWriteP0(UWB_RST_BIT, LOW);
    delay(20);

    Serial.println("[setupUWB] RST release (high-Z)");
    catSetPinModeP0(UWB_RST_BIT, false);   // release to pull-up
    delay(500);                            // important

    // 2) First communication test with explicit CS control + slow SPI
    uwbBeginAccess();

    // One-time SPI port setup (only the very first time).
    // spiBegin configures the IRQ pin and calls SPI.begin().
    // spiSelect MUST follow to set _ss in the library — without it, dwt_readdevid()
    // drives GPIO0 (uninitialized _ss=0) instead of UWB_SS.
    // Do NOT wrap dwt_readdevid() in a manual SPI.beginTransaction — the
library
    // calls it internally, and nesting two beginTransaction calls deadlocks the
    // ESP32 SPI semaphore.
    if (!uwbSpiPortReady) {
        spiBegin(UWB_IRQ, 0xFF);   // 0xFF = no lib-driven RST (we reset via
CAT9555)
        spiSelect(UWB_SS);         // sets _ss so readfromspi/writetospi use the right
CS
        uwbSpiPortReady = true;
    }

    Serial.printf("UWB_SS pin=%d  state before=%d\n", UWB_SS,
digitalRead(UWB_SS));

    // Let the library manage its own SPI transaction and CS toggling.
    uint32_t id = dwt_readdevid();
```

```
Serial.printf("DW ID after reset: 0x%08lX\n", id);
if (id == 0x00000000UL || id == 0xFFFFFFFFUL) {
    strncpy(uwbStatus, "UWB:no comm", sizeof(uwbStatus) - 1);
    uwbStatus[sizeof(uwbStatus) - 1] = '\0';
    Serial.println("[setupUWB] ERROR: no SPI response from DW3000");
    uwbEndAccess();
    return;
}

Serial.println("[setupUWB] DW3000 responded — good!");

// 3) Wait for IDLE_RC before soft reset
Serial.println("[setupUWB] Waiting for IDLE_RC");
int idleRetries = 100;
while (!dwt_checkidlerc()) {
    delay(10);
    if (--idleRetries <= 0) {
        strncpy(uwbStatus, "UWB:idle rc", sizeof(uwbStatus) - 1);
        Serial.println("[setupUWB] ERROR: IDLE_RC timeout before soft reset");
        uwbEndAccess();
        return;
    }
}

// 4) Soft reset
Serial.println("[setupUWB] Soft reset");
dwt_softreset();
delay(50);

// 5) Wait for IDLE_RC after soft reset
Serial.println("[setupUWB] Waiting for IDLE_RC after soft reset");
idleRetries = 100;
while (!dwt_checkidlerc()) {
    delay(10);
    if (--idleRetries <= 0) {
        strncpy(uwbStatus, "UWB:idle rc", sizeof(uwbStatus) - 1);
        Serial.println("[setupUWB] ERROR: IDLE_RC timeout after soft reset");
        uwbEndAccess();
        return;
    }
}

// 6) Library initialization
Serial.println("[setupUWB] Calling start_uwb()...");
uwbInitOk = start_uwb();
```

```
    Serial.print("[setupUWB] UWB init result: ");
    Serial.print(uwbInitOk ? "OK" : "FAIL");
    Serial.print("  last_error=");
    Serial.println(uwb_last_error);

    strncpy(uwbStatus, uwb_last_error, sizeof(uwbStatus) - 1);
    uwbStatus[sizeof(uwbStatus) - 1] = '\0';

    // Final cleanup
    uwbEndAccess();
}

void pollUWB() {
  if (!uwbActive || !uwbInitOk) return;
  static uint32_t lastPollMs = 0;
  static uint32_t lastRecoveryMs = 0;
  static uint16_t invalidPollCount = 0;
  if (!uwbForcePollPending && (millis() - lastPollMs <
UWB_POLL_INTERVAL_MS)) return;

// ---------------------------------------------------------------------
// Stale reading recovery — if no valid range has arrived in 500ms,
// force the DW3000 transceiver off and reset the ranging state machine.
// This clears any stuck RX/TX state that would block the next exchange.
// Rate-limited to once per 400ms to avoid thrashing the bus.
// ---------------------------------------------------------------------

    const uint32_t ageMs = millis() - uwbLatestUpdateMs;
    if (uwbLatestUpdateMs != 0 && ageMs > 500 && (millis() - lastRecoveryMs) >
400) {
      if (uwbTryBeginAccess()) {          // non-blocking — skip if LCD is using the
bus
        dwt_forcetrxoff();                // abort any in-progress TX or RX
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_ALL_RX_TO |
SYS_STATUS_ALL_RX_ERR);   // clear error flags
        counter = 0;                      // reset UWB state machine counter
        uwbEndAccess();
        lastRecoveryMs = millis();
        strncpy(uwbStatus, "UWB:recover", sizeof(uwbStatus) - 1);
        uwbStatus[sizeof(uwbStatus) - 1] = '\0';
        prevInsLine[7][0] = '\0';         // invalidate cached HUD line so it redraws
      }
    }
    // Reset distance to NAN before each attempt so stale values don't persist
    // if the initiator() call fails partway through
    distance = NAN;
```

```
  // Non-blocking mutex attempt — if LCD is mid-transaction, skip this poll
  // rather than blocking the render loop
  if (!uwbTryBeginAccess()) {
    return;
  }
  lastPollMs = millis();
  uwbForcePollPending = false;

  // Force transceiver off and clear all status flags before each fresh attempt.
  dwt_forcetrxoff();
  dwt_write32bitreg(SYS_STATUS_ID,
                    SYS_STATUS_TXFRS_BIT_MASK |  // TX frame sent
                    SYS_STATUS_RXFCG_BIT_MASK |  // RX frame good
                    SYS_STATUS_ALL_RX_TO |       // all RX timeout flags
                    SYS_STATUS_ALL_RX_ERR);      // all RX error flags
  counter = 0;   // reset ranging exchange state machine

  #ifdef INITIATOR
     initiator();   // sends poll, waits for response
  #else
     responder();   // waits for poll, sends response
  #endif
     uwbEndAccess(); // release SPI mutex

  //------------------Process Result----------------------------
 if (isfinite(distance) && distance > 0.05f && distance < 100.0f) {
   // Valid range received — latch into shared volatile variables
  uwbLatestDistanceM = (float)(distance);
  uwbLastGoodDistanceM = (float)(distance);
  uwbLatestUpdateMs = millis();
  uwbLatestValid = true;
  invalidPollCount = 0;
  snprintf(uwbStatus, sizeof(uwbStatus), "UWB:%5.2fm", (float)distance);
  prevInsLine[7][0] = '\0';  // force HUD redraw with new distance
  if (UWB_DEBUG && allowSerialDebug()) {
    Serial.print("UWB range: ");
    Serial.println((float)distance, 3);
  }
 } else { // No valid range this poll — show hold or polling status
  const uint32_t holdAgeMs = millis() - uwbLatestUpdateMs;
  if (uwbLatestValid && holdAgeMs < UWB_DISPLAY_HOLD_MS &&
isfinite(uwbLatestDistanceM)) {
     // Keep displaying last known distance for up to UWB_DISPLAY_HOLD_MS
    snprintf(uwbStatus, sizeof(uwbStatus), "UWB hold %5.2fm",
uwbLatestDistanceM);
  } else {  // Hold expired — mark invalid and show polling state
```

```
      uwbLatestValid = false;
      strncpy(uwbStatus, "UWB:polling", sizeof(uwbStatus) - 1);
      uwbStatus[sizeof(uwbStatus) - 1] = '\0';
    }
    if (invalidPollCount < 0xFFFF) invalidPollCount++;
    prevInsLine[7][0] = '\0';
    if (UWB_DEBUG && allowSerialDebug()) {
      Serial.println("UWB range invalid");
    }
  }
}

float insDistanceFromStartM() {
    if (!startSet) return NAN;
    // Project current position forward by one estimated step using velocity
    float projX = localX_m + vE * 0.005f;  // 5ms lookahead at typical poll rate
    float projY = localY_m + vN * 0.005f;
    float dx = projX - startX;
    float dy = projY - startY;
    return sqrtf(dx * dx + dy * dy);
}

void updateFusedDistance() {
    // ---------------------------------------------------------------------
    // Read UWB atomically — these are volatile and written from poll context
    // ---------------------------------------------------------------------
    float uwbM;
    uint32_t uwbMs;
    bool uwbHas;
    noInterrupts();
    uwbM = uwbLatestDistanceM;
    uwbMs = uwbLatestUpdateMs;
    uwbHas = uwbLatestValid;
    interrupts();

    float insM = insDistanceFromStartM();
    bool insValid = isfinite(insM);

    // UWB is only trusted if the reading is recent and finite
    uint32_t uwbAgeMs = millis() - uwbMs;
    bool uwbValid = uwbHas && isfinite(uwbM) && (uwbAgeMs < 1500);

    // ---------------------------------------------------------------------
    // UWB confidence — degrades as the reading ages beyond 200ms.
    // A reading that just arrived gets full weight; one that is 1.5s old
    // gets near-zero weight so INS takes over gracefully rather than snapping.
```

```
    // ---------------------------------------------------------------
    float uwbConfidence = 0.0f;
    if (uwbValid) {
        uwbConfidence = 1.0f - clampf((float)uwbAgeMs / 1500.0f, 0.0f, 1.0f);
        uwbConfidence = uwbConfidence * uwbConfidence;  // square for faster
rolloff
    }

    // ---------------------------------------------------------------
    // INS confidence — based on how long since setStartOrigin() and whether
    // the unit is currently moving. INS drift grows over time so confidence
    // decays slowly. Still detection resets the drift accumulator so we
    // give a small bonus when the unit is known stationary.
    // ---------------------------------------------------------------
    float insConfidence = 0.0f;
    if (insValid) {
        // Base confidence starts high and decays over ~5 minutes of DR
        static uint32_t lastOriginMs = 0;
        if (!startSet) {
            lastOriginMs = millis();
        }
        uint32_t drAgeMs = startSet ? (millis() - lastOriginMs) : 0;
        float drDecay = clampf(1.0f - (float)drAgeMs / (5.0f * 60.0f * 1000.0f), 0.15f,
1.0f);

        // Velocity magnitude — higher velocity means more confident INS is
tracking real motion
        float velMag = sqrtf(vE * vE + vN * vN);
        float motionBonus = clampf(velMag / 1.0f, 0.0f, 0.25f);

        // Still unit gets a bonus — drift is zero when stationary
        float stillBonus = (stillSampleCount >= STILL_CONFIRM_SAMPLES) ?
0.20f : 0.0f;

        insConfidence = clampf(drDecay + motionBonus + stillBonus, 0.0f, 1.0f);
    }

    // ---------------------------------------------------------------
    // Outlier rejection — if UWB and INS disagree by more than 3m and INS
    // has been running for less than 30 seconds (still reliable), treat the
    // UWB reading as a multipath artifact and reduce its weight heavily.
    // Beyond 30 seconds of DR, INS drift makes it unreliable so we trust UWB.
    // ---------------------------------------------------------------
    if (uwbValid && insValid) {
        float disagreement = fabsf(uwbM - insM);
        bool insRecent = (startSet && (millis() - insResetHoldUntilMs) < 30000UL);
```

```
    if (disagreement > 3.0f && insRecent) {
        uwbConfidence *= 0.30f;  // likely multipath — heavily discount UWB
    } else if (disagreement > 5.0f) {
        // Large disagreement at any time — trust neither fully, blend cautiously
        uwbConfidence *= 0.50f;
        insConfidence *= 0.50f;
    }
}

// ---------------------------------------------------------------------
// Compute weighted measurement
// ---------------------------------------------------------------------
float meas = NAN;
float totalConf = uwbConfidence + insConfidence;

if (totalConf > 0.01f) {
    float wUwb = uwbConfidence / totalConf;
    float wIns = insConfidence / totalConf;

    if (uwbValid && insValid) {
        meas = wUwb * uwbM + wIns * insM;
    } else if (uwbValid) {
        meas = uwbM;
    } else if (insValid) {
        meas = insM;
    }
} else if (uwbValid) {
    meas = uwbM;   // fallback if both confidences collapsed
} else if (insValid) {
    meas = insM;
}

if (!isfinite(meas)) return;

// ---------------------------------------------------------------------
// EMA fusion — tighter alpha when UWB is fresh and confident (fast track),
// looser when relying on INS only (slow smooth track to hide drift jumps).
// ---------------------------------------------------------------------
float fusionAlpha;
if (uwbValid && uwbConfidence > 0.70f) {
    fusionAlpha = 0.70f;   // UWB fresh — track it quickly
} else if (uwbValid) {
    fusionAlpha = 0.50f;   // UWB aging — moderate tracking
} else {
    fusionAlpha = 0.25f;   // INS only — smooth heavily to hide drift
}
```

```
    if (!isfinite(fusedDistanceM)) {
        fusedDistanceM = meas;   // seed on first valid measurement
    } else {
        fusedDistanceM = (1.0f - fusionAlpha) * fusedDistanceM + fusionAlpha *
meas;
    }
}
// Touch reads are another SPI transaction on the same wires, so LCD and UWB
// are explicitly deselected before sampling the controller.
// Reads one 12-bit touch-controller measurement over the shared SPI bus.
// This temporarily selects TOUCH_CS and uses a slower SPI clock suitable for
touch.

uint16_t touchRead12(uint8_t cmd) {
    if (spiBusMutex) xSemaphoreTake(spiBusMutex, portMAX_DELAY);
    deselectSharedSpiDevicesLocked();
    catDigitalWriteP0(TOUCH_CS_BIT, false);
    SPI.beginTransaction(SPISettings(500000, MSBFIRST, SPI_MODE0));
    SPI.transfer(cmd);
    uint16_t v = (SPI.transfer16(0x0000) >> 3) & 0x0FFF;
    SPI.endTransaction();
    catDigitalWriteP0(TOUCH_CS_BIT, true);
    if (spiBusMutex) xSemaphoreGive(spiBusMutex);
    return v;
}
void deselectSharedSpiDevicesLocked() {
    digitalWrite(TFT_CS, HIGH);
    digitalWrite(UWB_SS, HIGH);
    catDigitalWriteP0(TOUCH_CS_BIT, true);
}

void deselectSharedSpiDevices() {
    if (spiBusMutex) xSemaphoreTake(spiBusMutex, portMAX_DELAY);
    deselectSharedSpiDevicesLocked();
    if (spiBusMutex) xSemaphoreGive(spiBusMutex);
}
bool getTouchPoint(int &x, int &y) {
    if (currentMode == INS_MODE) return false;
    if (digitalRead(TOUCH_IRQ) == HIGH) return false;
    uint16_t z1 = touchRead12(0xB0);
    uint16_t z2 = touchRead12(0xC0);
    if (z1 < 120 || z2 > 3950) return false;
    uint32_t sx = 0, sy = 0;
    const int n = 7;
    for (int i = 0; i < n; i++) {
```

```
        sx += touchRead12(0xD0);
        sy += touchRead12(0x90);
    }
    uint16_t rx = sx / n;
    uint16_t ry = sy / n;
    int mx = map(rx, TOUCH_X_MIN, TOUCH_X_MAX, 0, 319);
    int my = map(ry, TOUCH_Y_MIN, TOUCH_Y_MAX, 0, 239);
    x = 319 - mx;
    y = 239 - my;
    x = constrain(x, 0, 319);
    y = constrain(y, 0, 239);
    return true;
}
static void invalidateScreenStateForCurrentMode() {
    topBarDirty = true;
    menuDrawn = false;
    if (currentMode == MENU) return;
    if (currentMode == THERMAL || currentMode == THERMAL_LOADING) {
        toolbarSelection = TOOLBAR_CLOSE;
        prevThermalValid = false;
        mlxInitDone = false;
        thermalHudInit = false;
        thermalHumanDetected = false;
        presenceScore = 0;
        return;
    }
    if (currentMode == PROXIMITY) {
        toolbarSelection = TOOLBAR_CLOSE;
        proxScreenInit = false;
        resetProximityReading();
        return;
    }
    if (currentMode == INS_MODE) {
        toolbarSelection = TOOLBAR_CLOSE;
        insScreenInit = false;
        insCacheInit = false;
        return;
    }
    if (currentMode == SCAN_MODE || currentMode == SCAN_LOADING) {
        toolbarSelection = TOOLBAR_CLOSE;
        scanScreenInit = false;
        scanReviewSequenceActive = false;
        scanReviewScreen = SCAN_REVIEW_SUMMARY;
        scanReviewCompletionLatched = false;
        scanThermalReviewInit = false;
        scanNavReviewInit = false;
```

```
        resetProximityReading();
        return;
    }
}
void enterScreensaver() {
    if (currentMode != MENU) return;
    if (audioState != AUDIO_IDLE && audioState != AUDIO_DONE) return;
    lcdBeginAccess();
    tft.fillScreen(ILI9341_BLACK);
    tft.startWrite();
    tft.setAddrWindow(0, 0, SAVER_W, SAVER_H);
    static uint16_t line[SAVER_W];
    for (int y = 0; y < SAVER_H; y++) {
        for (int x = 0; x < SAVER_W; x++) {
            line[x] = pgm_read_word(&epd_bitmap_2HADRADpic[y * SAVER_W +
x]);
        }
        tft.writePixels(line, SAVER_W);
    }
    tft.endWrite();
    lcdEndAccess();
    screensaverOn = true;
}
void exitScreensaver() {
    screensaverOn = false;
    invalidateScreenStateForCurrentMode();
    if (currentMode == MENU) {
        lcdBeginAccess();
        tft.fillScreen(ILI9341_BLACK);
        lcdEndAccess();
        drawMenu();
        playWelcomeIfIdle();
    }
}
void upscaleBilinear4x(const float *src, float *dst) {
    for (int uy = 0; uy < THERM_RENDER_H; uy++) {
        float fy = uy * (23.0f / (THERM_RENDER_H - 1));
        int y0 = (int)fy;
        int y1 = min(y0 + 1, 23);
        float wy = fy - y0;
        for (int ux = 0; ux < THERM_RENDER_W; ux++) {
            float fx = ux * (31.0f / (THERM_RENDER_W - 1));
            int x0 = (int)fx;
            int x1 = min(x0 + 1, 31);
            float wx = fx - x0;
            float a=src[y0*32+x0]+wx*(src[y0*32+x1]-src[y0*32+x0]);
```

```
            float b=src[y1*32+x0]+wx*(src[y1*32+x1]-src[y1*32+x0]);
            dst[uy * THERM_RENDER_W + ux] = a + wy * (b - a);
        }
    }
}
uint16_t thermalColor(float t, float lo, float hi) {
    float v = (hi > lo) ? ((t - lo) / (hi - lo)) : 0.5f;
    v = constrain(v, 0.0f, 1.0f);
    uint8_t r, g, b;
    if (v < 0.25f) {
        float u = v / 0.25f;
        r = 0;
        g = (uint8_t)(28 + 72 * u);
        b = (uint8_t)(96 + 135 * u);
    } else if (v < 0.50f) {
        float u = (v - 0.25f) / 0.25f;
        r = (uint8_t)(32 * u);
        g = (uint8_t)(116 + 88 * u);
        b = (uint8_t)(240 - 150 * u);
    } else if (v < 0.72f) {
        float u = (v - 0.50f) / 0.22f;
        r = (uint8_t)(255 * u);
        g = (uint8_t)(224 - 40 * u);
        b = 0;
    } else {
        float u = (v - 0.72f) / 0.28f;
        r = 255;
        g = (uint8_t)(180 - 180 * u);
        b = 0;
    }
    return tft.color565(r, g, b);
}
uint16_t thermalDisplayColor(float t, float ambient, float lo, float hi) {
    // Keep startup frames visually stable by mapping the filtered temperature
    // directly into the palette instead of aggressively boosting warm pixels.
    return thermalColor(t, lo, hi);
}
void analyzePresence(const float *img, float &ambient, float &maxT, int
&hotCount, bool &present) {
    static constexpr float HUMAN_CENTER_C = 33.5f;
    static constexpr float HUMAN_BAND_C = 3.0f;
    static constexpr float BG_ALPHA = 0.01f;
    static constexpr uint8_t HOLD_FRAMES = 2;
    static float bgModel[768];
    static bool bgInit = false;
    static uint8_t persistFrames = 0;
```

```
    static uint8_t visited[768];
    static int stack[768];
    float sum = 0.0f;
    maxT = -1000.0f;
    for (int i = 0; i < 768; i++) {
        sum += img[i];
        if (img[i] > maxT) maxT = img[i];
        if (!bgInit) bgModel[i] = img[i];
    }
    bgInit = true;
    ambient = sum / 768.0f;
    const float baseHumanMinC = HUMAN_CENTER_C - HUMAN_BAND_C;
    const float baseHumanMaxC = HUMAN_CENTER_C + HUMAN_BAND_C;
    const float adaptiveHumanMinC = max(29.0f, max(baseHumanMinC - 0.5f,
ambient + 0.35f));
    const float adaptiveHumanMaxC = min(40.0f, max(baseHumanMaxC, ambient
+ 4.5f));
    const float minBgDeltaC = (ambient >= 30.0f) ? 0.14f : 0.28f;
    const float minAmbientLiftC = (ambient >= 30.0f) ? 0.22f : 0.45f;
    hotCount = 0;
    bool hotMask[768];
    int minX = 31, minY = 23, maxX = 0, maxY = 0;
    int rowHot[24] = {0};
    int colHot[32] = {0};
    int hottestIdx = -1;
    for (int i = 0; i < 768; i++) {
        float t = img[i];
        float bgDelta = t - bgModel[i];
        bool hot =
            t > adaptiveHumanMinC &&
            t < adaptiveHumanMaxC &&
            t > ambient + minAmbientLiftC &&
            bgDelta > minBgDeltaC;
        hotMask[i] = hot;
        if (hot) {
            hotCount++;
            int x = i % 32;
            int y = i / 32;
            rowHot[y]++;
            colHot[x]++;
            if (x < minX) minX = x;
            if (x > maxX) maxX = x;
            if (y < minY) minY = y;
            if (y > maxY) maxY = y;
            if (hottestIdx < 0 || t > img[hottestIdx]) hottestIdx = i;
        } else {
```

```
            bgModel[i] += BG_ALPHA * (t - bgModel[i]);
        }
    }
    memset(visited, 0, sizeof(visited));
    int largestBlob = 0;
    for (int i = 0; i < 768; i++) {
        if (!hotMask[i] || visited[i]) continue;
        int sp = 0;
        int blobCount = 0;
        stack[sp++] = i;
        visited[i] = 1;
        while (sp > 0) {
            int idx = stack[--sp];
            blobCount++;
            int x = idx % 32;
            int y = idx / 32;
            const int neighbors[8][2] = {{1,0},{-1,0},{0,1},{0,-1},{1,1},{1,-1},{-1,1},{-1,-
1}};
            for (int n = 0; n < 8; n++) {
                int nx = x + neighbors[n][0];
                int ny = y + neighbors[n][1];
                if (nx < 0 || nx >= 32 || ny < 0 || ny >= 24) continue;
                int ni = ny * 32 + nx;
                if (!hotMask[ni] || visited[ni]) continue;
                visited[ni] = 1;
                stack[sp++] = ni;
            }
        }
        if (blobCount > largestBlob) largestBlob = blobCount;
    }
    int bboxW = (hotCount > 0) ? (maxX - minX + 1) : 0;
    int bboxH = (hotCount > 0) ? (maxY - minY + 1) : 0;
    int bboxArea = bboxW * bboxH;
    float aspect = (bboxH > 0) ? ((float)bboxW / (float)bboxH) : 0.0f;
    float compactness = (bboxArea > 0) ? ((float)largestBlob / (float)bboxArea) :
0.0f;
    bool headSizedBlob = (bboxW >= 2 && bboxW <= 10 && bboxH >= 2 &&
bboxH <= 10);
    bool nearFieldBlob = (bboxArea >= 45 || hotCount >= 40);
    bool farFieldBlob = (bboxArea >= 2 && bboxArea <= 24 && hotCount >= 2 &&
hotCount <= 18);
    bool standingProfile = (bboxH >= 4 && aspect >= 0.30f && aspect <= 1.90f);
    bool lyingProfile = (bboxW >= 4 && aspect >= 1.35f && aspect <= 2.8f);
    bool bodyProfile = standingProfile || lyingProfile;
    bool farFieldShape = (bboxW >= 1 && bboxW <= 8 && bboxH >= 2 && bboxH
<= 10 && aspect >= 0.35f && aspect <= 1.8f);
```

```
    bool headProfile = false;
    if (hotCount > 0) {
        int hottestX = hottestIdx % 32;
        int hottestY = hottestIdx / 32;
        int topRows = max(1, bboxH / 3);
        int topHot = 0;
        int topMinX = 31, topMaxX = 0, topMinY = 23, topMaxY = 0;
        for (int y = minY; y <= min(maxY, minY + topRows - 1); y++) {
            for (int x = minX; x <= maxX; x++) {
                int idx = y * 32 + x;
                if (!hotMask[idx]) continue;
                topHot++;
                if (x < topMinX) topMinX = x;
                if (x > topMaxX) topMaxX = x;
                if (y < topMinY) topMinY = y;
                if (y > topMaxY) topMaxY = y;
            }
        }
        int topW = (topHot > 0) ? (topMaxX - topMinX + 1) : 0;
        int topH = (topHot > 0) ? (topMaxY - topMinY + 1) : 0;
        float topAspect = (topH > 0) ? ((float)topW / (float)topH) : 0.0f;
        bool hottestNearTop = hottestY <= (minY + max(1, bboxH / 3));
        bool hottestNearCenter = hottestX >= (minX - 1 + bboxW / 4) && hottestX
<= (maxX + 1 - bboxW / 4);
        headProfile =
            topHot >= 2 &&
            topHot <= 28 &&
            topW >= 1 &&
            topH >= 1 &&
            topAspect >= 0.4f &&
            topAspect <= 2.2f &&
            hottestNearTop &&
            hottestNearCenter;
    }
    float maxLiftC = maxT - ambient;
    float blobDominance = (hotCount > 0) ? ((float)largestBlob / (float)hotCount) :
0.0f;
    int score = 0;
    if (maxT >= adaptiveHumanMinC && maxT <= adaptiveHumanMaxC) score
+= 18;
    if (maxLiftC >= minAmbientLiftC) score += 8;
    if (maxLiftC >= (minAmbientLiftC + 0.8f)) score += 8;
    if (hotCount >= 2 && hotCount <= 220) score += 10;
    if (largestBlob >= 2) score += 10;
    if (bboxArea >= 3 && bboxArea <= 220) score += 6;
    if (compactness >= 0.10f) score += 8;
```

```
    if (compactness >= 0.18f) score += 6;
    if (blobDominance >= 0.45f) score += 8;
    if (blobDominance >= 0.65f) score += 6;
    if (headProfile) score += 14;
    if (headProfile && headSizedBlob) score += 12;
    if (bodyProfile) score += 12;
    if (standingProfile) score += 5;
    if (lyingProfile) score += 4;
    if (farFieldShape) score += 10;
    if (farFieldBlob) score += 6;
    if (nearFieldBlob && compactness >= 0.18f) score += 12;
    thermalConfidencePct = constrain(score, 0, 100);
    bool headOnlyCandidate =
        headProfile &&
        headSizedBlob &&
        largestBlob >= 2 &&
        compactness >= 0.10f &&
        maxT >= adaptiveHumanMinC &&
        maxT <= adaptiveHumanMaxC;
    bool farFieldCandidate =
        farFieldBlob &&
        (headProfile || farFieldShape) &&
        largestBlob >= 2 &&
        compactness >= 0.10f &&
        maxT >= adaptiveHumanMinC &&
        maxT <= adaptiveHumanMaxC;
    bool nearFieldCandidate =
        nearFieldBlob &&
        largestBlob >= 6 &&
        compactness >= 0.18f &&
        maxT >= adaptiveHumanMinC &&
        maxT <= adaptiveHumanMaxC;
    bool candidate = thermalConfidencePct >= 58 || headOnlyCandidate ||
farFieldCandidate || nearFieldCandidate;
    if (candidate) {
        if (persistFrames < HOLD_FRAMES) persistFrames++;
    } else {
        persistFrames = 0;
    }
    bool instant = candidate && persistFrames >= HOLD_FRAMES;
    if (instant) presenceScore = min(30, presenceScore + 3);
    else presenceScore = max(0, presenceScore - 1);
    present = (presenceScore >= 8);
}
static float median3(float a, float b, float c) {
    if (a > b) { float t = a; a = b; b = t; }
```

```
    if (b > c) { float t = b; b = c; c = t; }
    if (a > b) { float t = a; a = b; b = t; }
    return b;
}

// ----------------USS-----------------------
static float readDistanceCmRaw() {
    // Trigger
    catDigitalWriteP0(USS_TRIG_BIT, true);
    delayMicroseconds(10);
    catDigitalWriteP0(USS_TRIG_BIT, false);

    // Wait for echo HIGH — tight timeout
    uint32_t t0 = micros();
    while (!catDigitalReadP0(USS_ECHO_BIT)) {
        if (micros() - t0 > 5000) return -1.0f;  // tighter — 5ms max wait
    }
    uint32_t t1 = micros();

    // Wait for echo LOW
    while (catDigitalReadP0(USS_ECHO_BIT)) {
        if (micros() - t1 > 12000) return -1.0f;  // 12ms = ~200cm
    }
    uint32_t dur = micros() - t1;

    float cm = dur * 0.0343f / 2.0f;
    if (cm < 2.0f || cm > 200.0f) return -1.0f;
    return cm;
}

float readDistanceCm() {
    float s0 = readDistanceCmRaw();
    delay(4);
    float s1 = readDistanceCmRaw();
    delay(4);
    float s2 = readDistanceCmRaw();
    float cm = -1.0f;
    if (s0 > 0.0f && s1 > 0.0f && s2 > 0.0f) cm = median3(s0, s1, s2);
    else if (s0 > 0.0f && s1 > 0.0f) cm = 0.5f * (s0 + s1);
    else if (s0 > 0.0f && s2 > 0.0f) cm = 0.5f * (s0 + s2);
    else if (s1 > 0.0f && s2 > 0.0f) cm = 0.5f * (s1 + s2);
    else if (s0 > 0.0f) cm = s0;
    else if (s1 > 0.0f) cm = s1;
    else if (s2 > 0.0f) cm = s2;
    if (cm <= 0.0f) return -1.0f;
    cm *= USS_CAL_GAIN;
```

```
    if (cm < 2.0f || cm > 200.0f) return -1.0f;   //final range check
    cm = roundf(cm / USS_DISPLAY_STEP_CM) * USS_DISPLAY_STEP_CM;
    return cm;
}
void resetProximityReading() {
    proxFiltered = -1.0f;
    proxTooCloseLatched = false;
    proxLastSampleMs = 0;
    proxInvalidCount = 0;
    proxCloseCount = 0;
}

void updateProximityReading() {
    if (millis() - proxLastSampleMs < 60) return;
    proxLastSampleMs = millis();
    float d = readDistanceCm();
    if (d >= 2.0f && d <= 200.0f) {  // cap at 200cm (2m)
        proxInvalidCount = 0;
        if (proxFiltered < 0.0f) proxFiltered = d;
        float limited = d;
        if (limited > proxFiltered + USS_MAX_STEP_CM) limited = proxFiltered +
USS_MAX_STEP_CM;
        if (limited < proxFiltered - USS_MAX_STEP_CM) limited = proxFiltered -
USS_MAX_STEP_CM;
        proxFiltered = 0.50f * proxFiltered + 0.50f * limited;
    } else {
        if (proxInvalidCount < 255) proxInvalidCount++;
        if (proxInvalidCount >= 3) {
            proxFiltered = -1.0f;
            proxCloseCount = 0;
        }
    }
    bool closeNow = (d > 0.0f && d < RADAR_STOP_THRESH_CM) ||
                    (proxFiltered > 0.0f && proxFiltered <
RADAR_STOP_THRESH_CM);
    if (closeNow) {
        if (proxCloseCount < 255) proxCloseCount++;
    } else {
        proxCloseCount = 0;
    }
    proxTooCloseLatched = closeNow;
}

// ----------------- RADAR INTERNAL STATE HELPERS ------------------
static void radarSetScanStatus(const char* status) {
    strncpy(radarScanStatus, status, sizeof(radarScanStatus) - 1);
```

```
    radarScanStatus[sizeof(radarScanStatus) - 1] = '\0';
}
static void radarResetLiveState() {
    radarHuman = false;
    radarTargetPresent = false;
    radarVitalsReady = false;
    radarBreathBpm = NAN;
    radarHeartBpm = NAN;
    radarHeartBpmDisplay  = NAN;
    radarBreathBpmDisplay = NAN;
    radarDistM = NAN;
    radarDistCandidateM = NAN;
    radarBreathCandidateBpm = NAN;
    radarHeartCandidateBpm = NAN;
    radarDistCandidateCount = 0;
    radarBreathCandidateCount = 0;
    radarHeartCandidateCount = 0;
    radarLastRangeBin = 0;
    radarLastPacketMs = millis();
    radarLastPresenceMs = 0;
    radarLastVitals1040Ms = 0;
    radarBufLen = 0;
    radarRangeProfileCount = 0;
    radarRangeProfileValid = false;
    radarRangeProfileMs = 0;
    radarPeakBin = -1;
    radarPeakScore = 0.0f;
    radarPeakCandidateCount = 0;
    radarPeakCandidateBin = -1;
    radarPeakRenderKey = 0;
    radarTlvStatCount = 0;
    radarLastTlvPrintMs = 0;
    strncpy(radarLastTlvSummary, "TLV:none", sizeof(radarLastTlvSummary) - 1);
    radarLastTlvSummary[sizeof(radarLastTlvSummary) - 1] = '\0';
    strncpy(radarDistanceHint, "D?:--", sizeof(radarDistanceHint) - 1);
    radarDistanceHint[sizeof(radarDistanceHint) - 1] = '\0';
    radarResetVitalMedians();
}
static void radarKillHard() {
    // Assert NRST LOW — immediately powers down the radar module
    pinMode(RADAR_NRST_PIN, OUTPUT);
    digitalWrite(RADAR_NRST_PIN, LOW);
    // Clean up serial and state
    RadarDataSerial.end();
    RadarSerial.end();
    radarEnabled = false;
```

```
    radarDataMode = false;
    radarResetLiveState();
    radarSetScanStatus("TOO CLOSE - radar off");
}

static bool radarWaitForCliPrompt(uint32_t timeoutMs) {
    String rx = "";
    unsigned long deadline = millis() + timeoutMs;
    while (millis() < deadline) {
        while (RadarSerial.available()) {
            char c = (char)RadarSerial.read();
            rx += c;
            if (rx.endsWith("mmwDemo:/>")) return true;
            if (rx.length() > 300) rx = rx.substring(150);
        }
        yield();
        delay(5);
    }
    return false;
}

static void radarSendConfigLines(const char* const cfg[]) {
    for (int i = 0; cfg[i]; i++) {
        RadarSerial.println(cfg[i]);
        const uint32_t delayMs = cfg[i + 1] ? RADAR_CMD_DELAY_MS :
RADAR_CMD_DELAY_FINAL_MS;
        delay(delayMs);
        yield();
    }
}

static void radarReviveFromKill() {
    // Release NRST — module begins booting
    pinMode(RADAR_NRST_PIN, OUTPUT);
    digitalWrite(RADAR_NRST_PIN, HIGH);
    delay(200);  // let module come out of reset before we talk to it

    if (!ensureRadarBuffer()) return;

    RadarSerial.begin(RADAR_CLI_BAUD, SERIAL_8N1, RADAR_RX_PIN,
RADAR_TX_PIN);
    while (RadarSerial.available()) RadarSerial.read();

    for (uint32_t t = 0; t < RADAR_BOOT_SETTLE_MS; t += 10) {
        delay(10);
        yield();
```

```
    }

    RadarSerial.print("\r\n");
    delay(40);
    radarWaitForCliPrompt(RADAR_PROMPT_TIMEOUT_MS);
    while (RadarSerial.available()) RadarSerial.read();
    radarSendConfigLines(radarCfgPhase2);

    RadarDataSerial.begin(RADAR_DATA_BAUD, SERIAL_8N1,
RADAR_DATA_RX_PIN, RADAR_DATA_TX_PIN);
    radarEnabled = true;
    radarDataMode = true;
    radarScanState = RADAR_SCAN_IDLE;
    radarResetLiveState();
    radarSetScanStatus("Proximity active");
}

static void radarResetScanResults() {
    radarPhase1HrSum = 0.0f;
    radarPhase1BrSum = 0.0f;
    radarPhase1HrAvg = NAN;
    radarPhase1BrAvg = NAN;
    radarPhase2DistAvg = NAN;
    radarPhase1SampleCount = 0;
    radarPhase2SampleCount = 0;
    radarPhaseStartMs = 0;
    radarBreathDisplayBpm = NAN;
    radarHeartDisplayBpm = NAN;
    radarDistDisplayM = NAN;
}
static void radarPushPhase2Sample(float sampleM, uint32_t nowMs) {
    if (!isfinite(sampleM) || sampleM <= 0.1f) return;
    if (radarPhase2SampleCount < RADAR_DISTANCE_SAMPLE_MAX) {
        radarPhase2Samples[radarPhase2SampleCount++] = { sampleM, nowMs };
        return;
    }
    memmove(radarPhase2Samples,
            radarPhase2Samples + 1,
            sizeof(radarPhase2Samples[0]) * (RADAR_DISTANCE_SAMPLE_MAX -
1));
    radarPhase2Samples[RADAR_DISTANCE_SAMPLE_MAX - 1] = { sampleM,
nowMs };
}
static float radarAveragePhase2Window(uint32_t cutoffMs) {
    float sum = 0.0f;
    uint16_t count = 0;
```

```
    for (uint16_t i = 0; i < radarPhase2SampleCount; i++) {
        if (radarPhase2Samples[i].tMs < cutoffMs) continue;
        if (!isfinite(radarPhase2Samples[i].valueM) || radarPhase2Samples[i].valueM
<= 0.1f) continue;
        sum += radarPhase2Samples[i].valueM;
        count++;
    }
    return count ? (sum / (float)count) : NAN;
}

const char* radarScanStateName() {
    switch (radarScanState) {
        case RADAR_SCAN_WAIT_P1:    return "Phase 1: waiting";
        case RADAR_SCAN_RUNNING_P1: return "Phase 1: HR/BR";
        case RADAR_SCAN_WAIT_P2:    return "Phase 2: waiting";
        case RADAR_SCAN_RUNNING_P2: return "Phase 2: distance";
        case RADAR_SCAN_DONE:       return "Complete";
        default:                    return "Idle";
    }
}

uint16_t radarScanProgressPermille() {
    switch (radarScanState) {
        case RADAR_SCAN_RUNNING_P1: {
            uint32_t elapsed = millis() - radarPhaseStartMs;
            return (uint16_t)min<uint32_t>(500, (elapsed * 500UL) /
RADAR_PHASE_MS);
        }
        case RADAR_SCAN_WAIT_P2:
            return 500;
        case RADAR_SCAN_RUNNING_P2: {
            uint32_t elapsed = millis() - radarPhaseStartMs;
            return (uint16_t)(500 + min<uint32_t>(500, (elapsed * 500UL) /
RADAR_PHASE_MS));
        }
        case RADAR_SCAN_DONE:
            return 1000;
        default:
            return 0;
    }
}

// Shared hardware-reset + config-send for both scan phases and proximity.
// Uses yield() inside the boot wait loops so the ESP32 watchdog is fed while
// the radar boots and accepts its config stream.
```

```
static bool radarStartPhaseConfig(const char* const cfg[], uint8_t waitState,
const char* status) {
   if (!ensureRadarBuffer()) {
      strncpy(radarLastTlvSummary, "RADAR:no mem",
sizeof(radarLastTlvSummary) - 1);
      radarLastTlvSummary[sizeof(radarLastTlvSummary) - 1] = '\0';
      radarSetScanStatus("Radar buffer failed");
      radarScanState = RADAR_SCAN_IDLE;
      return false;
   }
   RadarDataSerial.end();
   RadarSerial.end();
   RadarSerial.begin(RADAR_CLI_BAUD, SERIAL_8N1, RADAR_RX_PIN,
RADAR_TX_PIN);

   // Hardware reset on GPIO15 (RADAR_NRST_PIN).
   radarPulseReset();

   for (uint32_t t = 0; t < RADAR_BOOT_SETTLE_MS; t += 10) {
      delay(10);
      yield();
   }
   while (RadarSerial.available())    RadarSerial.read();
   while (RadarDataSerial.available()) RadarDataSerial.read();

   RadarSerial.print("\r\n");
   delay(40);
   radarWaitForCliPrompt(RADAR_PROMPT_TIMEOUT_MS);
   while (RadarSerial.available()) RadarSerial.read();
   radarSendConfigLines(cfg);

   RadarDataSerial.begin(RADAR_DATA_BAUD, SERIAL_8N1,
RADAR_DATA_RX_PIN, RADAR_DATA_TX_PIN);
   radarEnabled = true;
   radarDataMode = true;
   radarResetLiveState();
   radarScanState = (RadarScanState)waitState;
   radarPhaseStartMs = 0;
   radarSetScanStatus(status);
   return true;
}

static void radarCompletePhase2() {
   uint32_t cutoffMs = (radarPhaseStartMs + RADAR_PHASE_MS >
RADAR_LAST_DISTANCE_WINDOW_MS)
```

```
        ? (radarPhaseStartMs + RADAR_PHASE_MS -
RADAR_LAST_DISTANCE_WINDOW_MS)
        : 0;
    radarPhase2DistAvg = radarAveragePhase2Window(cutoffMs);
    radarDistDisplayM = radarPhase2DistAvg;
    radarScanState = RADAR_SCAN_DONE;
    radarSetScanStatus("Scan complete");
    // On the standalone radar (PROXIMITY) screen we want the radar to keep
    // streaming Phase 2 data (distance + HR/BR) after the two-phase sweep
    // ends so the live readouts and sonar dot keep updating. Only the
    // SCAN_MODE state machine needs the hardware stopped here.
    if (currentMode != PROXIMITY) {
        radarStop();
    }
}

static void radarCompletePhase1() {
    if (radarPhase1SampleCount > 0) {
        radarPhase1HrAvg = radarPhase1HrSum /
(float)radarPhase1SampleCount;
        radarPhase1BrAvg = radarPhase1BrSum /
(float)radarPhase1SampleCount;
        radarHeartDisplayBpm = radarPhase1HrAvg;
        radarBreathDisplayBpm = radarPhase1BrAvg;
    }
    radarPhase2SampleCount = 0;
    radarSetScanStatus("Phase 2: configuring");
    if (!radarStartPhaseConfig(radarCfgPhase2, RADAR_SCAN_WAIT_P2,
"Phase 2: waiting")) {
        radarScanState = RADAR_SCAN_IDLE;
    }
}

// Full two-phase scan. Called from SCAN tile START button.
void radarStart() {
    radarResetScanResults();
    if (!radarStartPhaseConfig(radarCfgPhase1, RADAR_SCAN_WAIT_P1,
"Phase 1: waiting")) {
        radarScanState = RADAR_SCAN_IDLE;
        return;
    }
    radarSetScanStatus("Phase 1: waiting");
}

// Stop radar and clean up. Preserves RADAR_SCAN_DONE state so the scan
// screen can continue displaying results after the sequence finishes.
```

```
void radarStop() {
  if (radarEnabled) {
    RadarSerial.updateBaudRate(RADAR_CLI_BAUD);
    RadarSerial.print("sensorStop\r\n");
    delay(40);
  }
  RadarDataSerial.end();
  RadarSerial.end();
  radarEnabled = false;
  radarDataMode = false;
  radarResetLiveState();
  releaseRadarBuffer();
  if (radarScanState != RADAR_SCAN_DONE) {
    radarScanState = RADAR_SCAN_IDLE;
    radarSetScanStatus("Press START");
  }
}

void radarProximityArmIdle() {
  stopAudioNow();
  if (radarEnabled) {
    RadarSerial.updateBaudRate(RADAR_CLI_BAUD);
    RadarSerial.print("sensorStop\r\n");
    delay(40);
  }
  RadarDataSerial.end();
  RadarSerial.end();
  radarInitResetPin();
  digitalWrite(RADAR_NRST_PIN, LOW);
  delay(5);
  radarEnabled = false;
  radarDataMode = false;
  radarScanState = RADAR_SCAN_IDLE;
  proxRadarHeldInReset = true;
  proxRadarRestartPrompt = false;
  radarResetLiveState();
  releaseRadarBuffer();
  radarSetScanStatus("Press START");
}

void radarStartProximity() {
  if (!ensureRadarBuffer()) return;
  proxScreenInit = false;
  proxRadarScopeBaseDrawn = false;

  if (radarEnabled) {
```

```
        radarSetScanStatus("Restarting radar...");
        RadarDataSerial.end();
        RadarSerial.end();
    }
    radarSetScanStatus("Starting radar...");
    RadarSerial.begin(RADAR_CLI_BAUD, SERIAL_8N1, RADAR_RX_PIN,
RADAR_TX_PIN);

    radarPulseReset();

    for (uint32_t t = 0; t < RADAR_PROX_BOOT_WAIT_MS; t += 10) { delay(10);
yield(); }
    while (RadarSerial.available()) RadarSerial.read();

    RadarSerial.print("\r\n");
    delay(40);
    radarWaitForCliPrompt(RADAR_PROX_PROMPT_TIMEOUT_MS);
    while (RadarSerial.available()) RadarSerial.read();

    for (int i = 0; radarCfgPhase2[i]; i++) {
        RadarSerial.println(radarCfgPhase2[i]);
        const uint32_t delayMs = radarCfgPhase2[i + 1] ?
RADAR_PROX_CMD_DELAY_MS : RADAR_CMD_DELAY_FINAL_MS;
        delay(delayMs);
        yield();
    }

    RadarDataSerial.begin(RADAR_DATA_BAUD, SERIAL_8N1,
RADAR_DATA_RX_PIN, RADAR_DATA_TX_PIN);
    radarEnabled = true;
    radarDataMode = true;
    radarScanState = RADAR_SCAN_IDLE;
    proxRadarHeldInReset = false;
    proxRadarRestartPrompt = false;
    radarResetLiveState();
    radarSetScanStatus("Proximity active");
}

void radarRecordTlv(uint32_t type, uint32_t tlvLen) {
    for (int i = 0; i < radarTlvStatCount; i++) {
        if (radarTlvStats[i].type == type) {
            radarTlvStats[i].count++;
            radarTlvStats[i].lastLen = tlvLen;
            return;
        }
    }
```

```
    if (radarTlvStatCount < RADAR_TLV_STAT_MAX) {
        radarTlvStats[radarTlvStatCount++] = { type, 1, tlvLen };
    }
}
void radarPrintTlvStats() {
  if (millis() - radarLastTlvPrintMs < RADAR_TLV_LOG_PERIOD_MS) return;
  radarLastTlvPrintMs = millis();
}
void radarDumpTlvSample(uint32_t type, const uint8_t* payload, uint32_t tlvLen)
{
  uint32_t nowMs = millis();
  if (nowMs - radarLastTlvDumpMs < RADAR_TLV_LOG_PERIOD_MS &&
      type == radarLastDumpedType &&
      tlvLen == radarLastDumpedLen) {
    return;
  }
  radarLastTlvDumpMs = nowMs;
  radarLastDumpedType = type;
  radarLastDumpedLen = tlvLen;
}
void radarReportKnownMapTlv(const char* label, uint32_t tlvLen) {
  static uint32_t lastReportMs = 0;
    static uint32_t lastReportHash = 0;
    uint32_t nowMs = millis();
    uint32_t hash = tlvLen;
    for (const char* p = label; *p; ++p) hash = hash * 33u + (uint8_t)(*p);
  if (nowMs - lastReportMs < RADAR_TLV_LOG_PERIOD_MS && hash ==
lastReportHash) return;
  lastReportMs = nowMs;
  lastReportHash = hash;
}
void radarUpdateRangeProfile(const uint8_t* payload, uint32_t tlvLen) {
    if ((tlvLen & 1u) != 0u) return;
    int count = min<int>((int)(tlvLen / 2u), RADAR_PROFILE_MAX_BINS);
    if (count <= 0) return;
    for (int i = 0; i < count; i++) {
        radarRangeProfile[i] = rdU16(payload + i * 2);
    }
    radarRangeProfileCount = count;
    radarRangeProfileValid = true;
    radarRangeProfileMs = millis();
}

void radarParseFrame(const uint8_t* frame, int len) {
    if (len < 40) return;
    uint32_t totalLen = rdU32(frame + 12);
```

```
    uint32_t numTlvs = rdU32(frame + 32);
    if (totalLen < 40 || totalLen > (uint32_t)len || numTlvs > 64) return;
    radarLastPacketMs = millis();
    int idx = 40;
    for (uint32_t t = 0; t < numTlvs; t++) {
      if (idx + 8 > (int)totalLen) break;
      uint32_t type = rdU32(frame + idx);
      uint32_t tlvLen = rdU32(frame + idx + 4);
      uint32_t pay = (uint32_t)idx + 8u;
      if (pay > totalLen || tlvLen > (totalLen - pay)) break;
      radarRecordTlv(type, tlvLen);
      snprintf(radarLastTlvSummary, sizeof(radarLastTlvSummary), "TLV:%lu
L:%lu",
               (unsigned long)type, (unsigned long)tlvLen);

      if (type == RADAR_TLV_RANGE_PROFILE) {
        radarUpdateRangeProfile(frame + pay, tlvLen);
        snprintf(radarLastTlvSummary, sizeof(radarLastTlvSummary), "RP
bins:%lu",
                 (unsigned long)(tlvLen / 2u));

      } else if (type == RADAR_TLV_PRESENCE && tlvLen >= 4) {
        if (rdU32(frame + pay) == 1) {
          radarTargetPresent = true;
          radarLastPresenceMs = millis();
        }

      } else if (type == RADAR_TLV_TARGET_LIST && tlvLen >= 12) {
        // Y-axis position from the target list is the forward distance.
        float posY = rdF32(frame + pay + 8);
        if (isfinite(posY) && posY > 0.1f && posY < 8.0f) {
          radarUpdateDistance(posY, true);
          snprintf(radarDistanceHint, sizeof(radarDistanceHint), "D:%.2fm",
posY);
          if (radarScanState == RADAR_SCAN_RUNNING_P2) {
            radarPushPhase2Sample(posY, millis());
          }
        }

      } else if (type == RADAR_TLV_VITALSIGNS && tlvLen >=
RADAR_TLV_VITALSIGNS_LEN) {
        radarLastVitals1040Ms = millis();
        // TI vital-signs TLV layout (type 1040):
        //   u16 unique_id     @ 0
        //   u16 range_bin     @ 2   <-- we want this for distance
        //   f32 breath_dev    @ 4
```

```
        //   f32 heart_rate    @ 8
        //   f32 breath_rate   @ 12
        //   f32 waveform[30]  @ 16
        // The scan Phase-1 config does not emit TLV 1010 (target list),
        // so the range_bin carried in 1040 is our only live distance
        // source while the scan radar phase is running.
        uint16_t rangeBin     = rdU16(frame + pay + 2);
        float breathDeviation = rdF32(frame + pay + 4);
        float heartRate       = rdF32(frame + pay + 8);
        float breathRate      = rdF32(frame + pay + 12);

        if (rangeBin > 0) {
            radarLastRangeBin = rangeBin;
            float rangeM = radarRangeBinToMeters(rangeBin);
            // Treat the vitals-TLV bin as non-authoritative so if TLV 1010
            // ever does come in (richer tracker output) it still wins the
            // EMA via radarUpdateDistance(..., true).
            radarUpdateDistance(rangeM, false);
        }
        bool hrValid = isfinite(heartRate)  && heartRate  > 40.0f && heartRate  <
150.0f
                    && fabsf(heartRate - 50.2f) > 1.0f;
        bool brValid = isfinite(breathRate) && breathRate > 5.0f  && breathRate <
40.0f;

        if (fabsf(breathDeviation) > 0.02f) {
            radarTargetPresent = true;
            radarLastPresenceMs = millis();
        }
        // Pass each valid sample through the median filter before it's
        // allowed anywhere else. Spike samples (common with the current
        // vitals-only config) get suppressed, and the downstream radar
        // HR / BR variables are always the median of the last ~1 s.
        float hrFiltered = NAN;
        float brFiltered = NAN;
        if (hrValid) {
            hrFiltered = radarPushHrSample(heartRate);
            if (isfinite(hrFiltered)) radarHeartBpm = hrFiltered;
        }
        if (brValid) {
            brFiltered = radarPushBrSample(breathRate);
            if (isfinite(brFiltered)) radarBreathBpm = brFiltered;
        }
        radarVitalsReady = isfinite(radarHeartBpm) && isfinite(radarBreathBpm);

        // Feed the display-side EMAs used by the radar (proximity) screen.
```

```
            // This is the "radar-screen equivalent" of the scan screen's
            // phase-1 30-second running average: it stacks on top of the
            // 1-second median already baked into radarHeartBpm / radarBreathBpm
            // and gives the live radar screen a calm, readable HR/BR without
            // requiring a phase-1 window.
            if (isfinite(radarHeartBpm)) {
                if (!isfinite(radarHeartBpmDisplay)) radarHeartBpmDisplay =
radarHeartBpm;
                else radarHeartBpmDisplay += RADAR_DISPLAY_EMA_ALPHA_HR *
                                        (radarHeartBpm - radarHeartBpmDisplay);
            }
            if (isfinite(radarBreathBpm)) {
                if (!isfinite(radarBreathBpmDisplay)) radarBreathBpmDisplay =
radarBreathBpm;
                else radarBreathBpmDisplay += RADAR_DISPLAY_EMA_ALPHA_BR
*
                                         (radarBreathBpm - radarBreathBpmDisplay);
            }

            // Accumulate running averages during Phase 1 using the filtered
            // samples so a few outliers early in the phase no longer drag the
            // 30-second average off-target.
            if (radarScanState == RADAR_SCAN_RUNNING_P1 &&
                isfinite(hrFiltered) && isfinite(brFiltered)) {
                radarPhase1HrSum += hrFiltered;
                radarPhase1BrSum += brFiltered;
                radarPhase1SampleCount++;
                radarHeartDisplayBpm  = radarPhase1HrSum /
(float)radarPhase1SampleCount;
                radarBreathDisplayBpm = radarPhase1BrSum /
(float)radarPhase1SampleCount;
            }
            snprintf(radarLastTlvSummary, sizeof(radarLastTlvSummary), "VS rb:%u
hr:%.1f br:%.1f",
                     (unsigned)radarLastRangeBin, heartRate, breathRate);

        } else if (type == RADAR_TLV_DETECTED_POINTS_COMPRESSED) {
            radarReportKnownMapTlv("TYPE301", tlvLen);
        } else if (type == RADAR_TLV_RANGE_AZIMUTH_HEATMAP_MAJOR) {
            radarReportKnownMapTlv("TYPE304", tlvLen);
        } else if (type == RADAR_TLV_RANGE_AZIMUTH_HEATMAP_MINOR) {
            radarReportKnownMapTlv("TYPE305", tlvLen);
        } else if (type == RADAR_TLV_RANGE_DOPPLER_HEATMAP) {
            radarReportKnownMapTlv("TYPE5", tlvLen);
        } else {
            radarDumpTlvSample(type, frame + pay, tlvLen);
```

```
        }
        idx += 8 + (int)tlvLen;
    }
    radarPrintTlvStats();
}

// Drain the radar data UART, realign on the mmWave magic word, parse each
// full frame, and drive the two-phase state machine transitions.
void radarPoll() {
    if (!radarEnabled || !radarDataMode || !radarBuf) return;
    while (RadarDataSerial.available() && radarBufLen < RADAR_BUF_SIZE) {
        radarBuf[radarBufLen++] = (uint8_t)RadarDataSerial.read();
    }
    while (radarBufLen >= 40) {
        if (memcmp(radarBuf, RADAR_MAGIC, 8) != 0) {
            memmove(radarBuf, radarBuf + 1, --radarBufLen);
            continue;
        }
        uint32_t totalLen = rdU32(radarBuf + 12);
        uint32_t numTlvs = rdU32(radarBuf + 32);
        if (totalLen < 40 || totalLen > RADAR_MAX_FRAME_LEN || numTlvs == 0 ||
numTlvs > 20) {
            memmove(radarBuf, radarBuf + 1, --radarBufLen);
            continue;
        }
        if (radarBufLen < (int)totalLen) break;
        radarParseFrame(radarBuf, (int)totalLen);
        radarBufLen -= (int)totalLen;
        memmove(radarBuf, radarBuf + totalLen, radarBufLen);
    }

    uint32_t nowMs = millis();

    // Presence timeout: if nothing heard for 10 s, clear detection state.
    if (radarLastPresenceMs > 0 && nowMs - radarLastPresenceMs >
RADAR_PRESENCE_TIMEOUT_MS) {
        radarTargetPresent = false;
        radarHuman = false;
        radarVitalsReady = false;
        radarBreathBpm = NAN;
        radarHeartBpm = NAN;
        radarDistM = NAN;
        strncpy(radarDistanceHint, "D?:--", sizeof(radarDistanceHint) - 1);
        radarDistanceHint[sizeof(radarDistanceHint) - 1] = '\0';
    } else {
        radarHuman = radarTargetPresent && radarVitalsReady;
```

```
    }

    // Phase state machine transitions.
    if (radarScanState == RADAR_SCAN_WAIT_P1 && radarHuman) {
        radarPhaseStartMs = nowMs;
        radarPhase1HrSum = 0.0f;
        radarPhase1BrSum = 0.0f;
        radarPhase1SampleCount = 0;
        radarHeartDisplayBpm = NAN;
        radarBreathDisplayBpm = NAN;
        radarScanState = RADAR_SCAN_RUNNING_P1;
        radarSetScanStatus("Phase 1: collecting");

    } else if (radarScanState == RADAR_SCAN_RUNNING_P1) {
        if (nowMs - radarPhaseStartMs >= RADAR_PHASE_MS) {
            radarCompletePhase1();
            return;
        }

    } else if (radarScanState == RADAR_SCAN_WAIT_P2 && radarHuman) {
        radarPhaseStartMs = nowMs;
        radarPhase2SampleCount = 0;
        radarDistDisplayM = NAN;
        radarScanState = RADAR_SCAN_RUNNING_P2;
        radarSetScanStatus("Phase 2: collecting");

    } else if (radarScanState == RADAR_SCAN_RUNNING_P2) {
        // Keep a rolling display value during the phase.
        uint32_t rollingCutoff = (nowMs >
RADAR_LAST_DISTANCE_WINDOW_MS)
            ? (nowMs - RADAR_LAST_DISTANCE_WINDOW_MS) : 0;
        radarDistDisplayM = radarAveragePhase2Window(rollingCutoff);
        if (nowMs - radarPhaseStartMs >= RADAR_PHASE_MS) {
            radarCompletePhase2();
            return;
        }
    }
}

// Legacy helper kept for any code path that still needs it.
bool radarWaitForPrompt() {
    String rx = "";
    unsigned long deadline = millis() + 10000;
    while (millis() < deadline) {
        while (RadarSerial.available()) {
            char c = (char)RadarSerial.read();
```

```
            rx += c;
            if (rx.endsWith("mmwDemo:/>")) return true;
            if (rx.length() > 200) rx = rx.substring(100);
        }
    }
    return false;
}

// ----------------- INS UPDATE PIPELINE -----------------
// INS updates run continuously while the INS screen is active. The raw local
// solution is allowed to drift internally, while displayX/displayY can stay
// steadier so the ASCII map does not slide when the unit is stationary.
void addTrackPoint(float x, float y) {
    track[trackHead] = {x, y};
    trackHead = (trackHead + 1) % TRACK_MAX;
    if (trackCount < TRACK_MAX) trackCount++;
}
void clearTrack() {
    trackCount = 0;
    trackHead = 0;
}
void clearVictims() {
    victimCount = 0;
    victimLogCount = 0;
    victimLogHead = 0;
}
void pushVictimSnapshot() {
    VictimSnapshot s;
    float ox = startSet ? startX : 0.0f;
    float oy = startSet ? startY : 0.0f;
    s.e_m = localX_m - ox;
    s.n_m = localY_m - oy;
    s.yaw_deg = yawDeg;
    s.alt_m = (isfinite(altitudeM) && isfinite(altitudeZeroM)) ? (altitudeM -
altitudeZeroM) : altitudeM;
    s.temp_c = temperatureC;
    s.pressure_hpa = pressureHpa;
    s.gpsFix = gpsValidNow;
    s.lat = lastLat;
    s.lon = lastLon;
    s.tMs = millis();
    victimLog[victimLogHead] = s;
    victimLogHead = (victimLogHead + 1) % VICTIM_LOG_MAX;
    if (victimLogCount < VICTIM_LOG_MAX) victimLogCount++;
}
void markVictim() {
```

```
    if (!startSet) return;
    uwbActive = true;
    uwbForcePollPending = true;
    if (isfinite(uwbLatestDistanceM)) {
        uwbLatestValid = true;
        uwbLatestUpdateMs = millis();
    }
    if (victimCount < VICTIM_MAX) {
        victims[victimCount++] = {localX_m, localY_m};
    } else {
        for (int i = 1; i < VICTIM_MAX; i++) {
            victims[i - 1] = victims[i];
        }
        victims[VICTIM_MAX - 1] = {localX_m, localY_m};
    }
    pushVictimSnapshot();
    prevInsLine[4][0] = '\0';
    prevInsLine[5][0] = '\0';
    prevInsLine[8][0] = '\0';
    for (int r = 0; r < MAP_H; r++) {
        prevInsGrid[r][0] = '\0';
    }
}
void calibrateIMUQuick() {
    const int N = 300;
    float sax = 0, say = 0, saz = 0, sgz = 0;
    int n = 0;
    for (int i = 0; i < N; i++) {
        pumpGNSS();
        if (mpu.update()) {
            sax += mpu.getAccX() * G;
            say += mpu.getAccY() * G;
            saz += mpu.getAccZ() * G;
            sgz += mpu.getGyroZ();
            n++;
        }
        delay(4);
    }
    if (n > 0) {
        accBiasX = sax / n;
        accBiasY = say / n;
        accBiasZ = (saz / n) - G;
        gyroZBias = sgz / n;
    }
}
void calibrateAltitudeBaseline() {
```

```
    if (!bmpReady) return;
    const int sampleCount = 12;
    float sumAlt = 0.0f;
    int validCount = 0;
    for (int i = 0; i < sampleCount; i++) {
        float p = bmp.readPressure() / 100.0f;
        if (isfinite(p) && p > 850.0f && p < 1100.0f) {
            float alt = 44330.0f * (1.0f - powf(p / SEA_LEVEL_HPA, 0.19029495f));
            if (isfinite(alt)) {
                sumAlt += alt;
                validCount++;
            }
        }
        delay(20);
    }
    if (validCount > 0) altitudeZeroM = sumAlt / validCount;
}
void setStartOrigin() {
    calibrateIMUQuick();
    calibrateAltitudeBaseline();
    vE = 0.0f; vN = 0.0f;
    aELpf = 0.0f; aNLpf = 0.0f;
    stillSampleCount = 0;
    yawDeg = 0.0f;
    yawInit = true;
    insPrevUs = micros();
    yawZero = yawDeg;
    localX_m = 0.0f;
    localY_m = 0.0f;
    displayX_m = 0.0f;
    displayY_m = 0.0f;
    localOriginSet = false;
    insResetHoldUntilMs = millis() + 1200;
    lastInsTrackMs = millis();
    startX = localX_m;
    startY = localY_m;
    startSet = true;
    clearTrack();
    float lastTrackX, lastTrackY;
    lastTrackX = 0.0f;
    lastTrackY = 0.0f;
    clearVictims();
    addTrackPoint(localX_m, localY_m);
}

void updateINS() {
```

```
    // Drain any buffered GNSS bytes into the TinyGPS parser — called every INS update
    // so GPS fix state stays current even when the IMU update rate is the bottleneck
    pumpGNSS();
    if (!imuReady) return;
    if (!mpu.update()) return;   // MPU9250 update returns false if no new data is available yet

    // Compute elapsed time since last update in seconds for integration
    uint32_t nowUs = micros();
    if (insPrevUs == 0) insPrevUs = nowUs;
    float dt = (nowUs - insPrevUs) * 1e-6f;
    insPrevUs = nowUs;
    if (dt <= 0.0f) return;
    if (dt > 0.20f) dt = 0.20f;   // Cap dt to 200ms — protects against large jumps after the INS screen was idle

    // Low-pass filter the dt to get a stable sample rate estimate for the Madgwick filter
    static float dtFilt = 0.005f;
    dtFilt = 0.90f * dtFilt + 0.10f * dt;

    // Read accelerometer in m/s^2 and subtract per-axis bias calibrated at setStartOrigin()
    float ax = (mpu.getAccX() * G) - accBiasX;
    float ay = (mpu.getAccY() * G) - accBiasY;
    float az = (mpu.getAccZ() * G) - accBiasZ;

    // Update compass heading from magnetometer — NAN if mag data is invalid
    compassHeadingDeg = computeCompassHeadingDeg();
    compassValid = isfinite(compassHeadingDeg);

    // Read gyroscope in degrees/sec — subtract Z-axis bias for yaw integration
    float gx = mpu.getGyroX();
    float gy = mpu.getGyroY();
    float gz = mpu.getGyroZ() - gyroZBias;

    // Feed gyro and accelerometer into the Madgwick AHRS filter
    // This computes a quaternion orientation estimate fusing both sensors
    float sampleHz = constrain(1.0f / dtFilt, 5.0f, 500.0f);
    filter.begin(sampleHz);
    filter.updateIMU(gx, gy, gz, ax / G, ay / G, az / G);

    // Extract Euler angles from the Madgwick filter output
    float roll = filter.getRoll() * DEG_TO_RAD;
```

```
  float pitchDegNow = correctPitchDegrees(filter.getPitch());
  float pitch = pitchDegNow * DEG_TO_RAD;
  float rollDegNow = roll * RAD_TO_DEG;
  rollDegNowAbs = rollDegNow;
  pitchDegNowAbs = pitchDegNow;

  // Integrate gyro Z to maintain our own yaw — Madgwick pitch/roll are used
  // for tilt compensation but its yaw drifts too fast for indoor navigation
  if (!yawInit) { yawDeg = 0; yawInit = true; }
  yawDeg = wrap360(yawDeg + gz * dt);

  // Build the 3x3 rotation matrix from roll/pitch/yaw
  // Used to rotate body frame acceleration into the world (East/North) frame
  float cy = cosf(yawDeg * DEG_TO_RAD), sy = sinf(yawDeg * DEG_TO_RAD);
  float cp = cosf(pitch), sp = sinf(pitch);
  float cr = cosf(roll), sr = sinf(roll);
  // R11..R23 are the first two rows of the rotation matrix (East and North axes)
  float R11 = cy * cp;
  float R12 = cy * sp * sr - sy * cr;
  float R13 = cy * sp * cr + sy * sr;
  float R21 = sy * cp;
  float R22 = sy * sp * sr + cy * cr;
  float R23 = sy * sp * cr - cy * sr;

  // Rotate body-frame acceleration into world East/North components
  float aE = R11 * ax + R12 * ay + R13 * az;
  float aN = R21 * ax + R22 * ay + R23 * az;

  // Update GPS fix state and cache last valid coordinates
  gpsValidNow = gps.location.isValid();
  if (gpsValidNow) {
    lastLat = gps.location.lat();
    lastLon = gps.location.lng();
  }
  if (millis() < insResetHoldUntilMs) {
    // Freeze position and velocity for ~1.2s after setStartOrigin() so the
    // IMU calibration bias subtraction can settle before we start integrating
    vE = 0.0f; vN = 0.0f;
    aELpf = 0.0f; aNLpf = 0.0f;
    localX_m = 0.0f; localY_m = 0.0f;
  } else {
    // Low-pass filter the world-frame acceleration to reduce high-frequency
noise
    aELpf = AXY_LPF * aE + (1.0f - AXY_LPF) * aELpf;
    aNLpf = AXY_LPF * aN + (1.0f - AXY_LPF) * aNLpf;
    // Apply deadband — zero out tiny accelerations that are likely bias residual
```

```
        if (fabsf(aELpf) < AXY_DEADBAND) aELpf = 0;
        if (fabsf(aNLpf) < AXY_DEADBAND) aNLpf = 0;

        // Still detection — check if both accelerometer and gyro indicate the unit is
stationary
        float accMagG = sqrtf((ax * ax) + (ay * ay) + (az * az)) / G;
        float gyroMagDps = sqrtf((gx * gx) + (gy * gy) + (gz * gz));
        bool looksStill =
            fabsf(accMagG - 1.0f) < STILL_ACC_BAND &&    // total acc within band
of 1g (gravity only)
            gyroMagDps < STILL_GYRO_DPS &&               // rotation rate below
threshold
            fabsf(aELpf) < (AXY_DEADBAND * 1.5f) &&      // filtered horizontal acc
negligible
            fabsf(aNLpf) < (AXY_DEADBAND * 1.5f);

        // Require STILL_CONFIRM_SAMPLES consecutive still frames before
zeroing velocity
        if (looksStill) {
            if (stillSampleCount < STILL_CONFIRM_SAMPLES) stillSampleCount++;
        } else {
            stillSampleCount = 0;
        }

        if (stillSampleCount >= STILL_CONFIRM_SAMPLES) { // Unit confirmed
stationary — zero velocity and acceleration to prevent drift
            vE = 0.0f; vN = 0.0f;
            aELpf = 0.0f; aNLpf = 0.0f;
        } else {
            // Integrate acceleration into velocity, then velocity into position
            // VEL_DAMP slightly reduces velocity each step to model real-world
friction
            // and prevent unbounded drift when still detection is slow to trigger
            vE = (vE + aELpf * dt) * VEL_DAMP;
            vN = (vN + aNLpf * dt) * VEL_DAMP;
             // Clamp velocity to a realistic walking/running maximum
            vE = clampf(vE, -VEL_MAX, VEL_MAX);
            vN = clampf(vN, -VEL_MAX, VEL_MAX);
            // Dead-reckon position by integrating velocity
            localX_m += vE * dt;
            localY_m += vN * dt;
        }
    }

    // Sample barometer if available — reads temperature and pressure then
converts to altitude
```

```
    if (bmpReady) {
        float t = bmp.readTemperature();
        float p = bmp.readPressure() / 100.0f;   // Pa to hPa
        if (isfinite(t) && isfinite(p) && p > 850 && p < 1100) {
            temperatureC = t;
            float correctedAmbientC = t - BMP_TEMP_DISPLAY_OFFSET_C;
            if (!isfinite(insAmbientDisplayC)) insAmbientDisplayC =
correctedAmbientC;
            else insAmbientDisplayC = emaUpdate(insAmbientDisplayC,
correctedAmbientC, 0.10f);
            pressureHpa = p;
            altitudeM = 44330.0f * (1.0f - powf(p / SEA_LEVEL_HPA, 0.19029495f));
// Altitude from pressure and sea-level reference
        }
    }

    uint32_t nowMs = millis();
    if (startSet) {
        static float lastTrackX = 0.0f;
        static float lastTrackY = 0.0f;
        float dx = localX_m - lastTrackX;
        float dy = localY_m - lastTrackY;
        float distSinceLastPoint = sqrtf(dx * dx + dy * dy);
        if (distSinceLastPoint >= 2.0f) {
            lastInsTrackMs = nowMs;
            addTrackPoint(localX_m, localY_m);
            lastTrackX = localX_m;
            lastTrackY = localY_m;
        }
    }

    float displayDx = localX_m - displayX_m;
    float displayDy = localY_m - displayY_m;
    bool movingNow =
        stillSampleCount < STILL_CONFIRM_SAMPLES &&
        (fabsf(vE) > 0.05f || fabsf(vN) > 0.05f ||
        fabsf(displayDx) > DISPLAY_MOVE_THRESH_M ||
        fabsf(displayDy) > DISPLAY_MOVE_THRESH_M);
    if (movingNow || !startSet) {
        displayX_m = localX_m;
        displayY_m = localY_m;
    }

    rollDegDisp = rollDegNow - rollZero;
    pitchDegDisp = pitchDegNow - pitchZero;
    yawDegDisp = yawDeg - yawZero;
```

```
    if (yawDegDisp > 359.0f || yawDegDisp < 1.0f) yawDegDisp = 0.0f;
    if (yawDegDisp < 0) yawDegDisp += 360.0f;
    if (yawDegDisp >= 360.0f) yawDegDisp -= 360.0f;
    if (fabsf(rollDegDisp) < 0.2f) rollDegDisp = 0.0f;
    if (fabsf(pitchDegDisp) < 0.2f) pitchDegDisp = 0.0f;
    }

// Compass rendering is driven from the magnetometer heading
    // Compute tilt-compensated compass heading in degrees (0-360) from the
// MPU9250 magnetometer. Uses the current roll and pitch from the Madgwick
// filter to rotate the magnetometer vector into the horizontal plane before
// computing the heading angle. This removes the error introduced when the
// unit is tilted — without this correction a 30° tilt can cause 10-20°
// of heading error depending on magnetic field strength at your location.
float computeCompassHeadingDeg() {
    float mx = mpu.getMagX();
    float my = mpu.getMagY();
    float mz = mpu.getMagZ();

    // Reject invalid readings
    if (!isfinite(mx) || !isfinite(my) || !isfinite(mz)) return NAN;

    // Reject near-zero readings — saturated or uninitialized sensor
    if (fabsf(mx) < 1e-4f && fabsf(my) < 1e-4f && fabsf(mz) < 1e-4f) return NAN;

    // Get current roll and pitch from the Madgwick filter in radians
    // These are the same values used by updateINS() for position integration
    float roll  = filter.getRoll()  * DEG_TO_RAD;
    float pitch = filter.getPitch() * DEG_TO_RAD;

    float cr = cosf(roll);
    float sr = sinf(roll);
    float cp = cosf(pitch);
    float sp = sinf(pitch);

    // Rotate the magnetometer vector into the horizontal plane using the
    // roll-pitch rotation matrix. This is a two-step rotation:
    // 1. Correct for pitch (rotation around Y axis)
    // 2. Correct for roll  (rotation around X axis)
    // The result is what the magnetometer would read if the unit were level.
    float mxH =  mx * cp + my * sr * sp + mz * cr * sp;
    float myH =            my * cr      - mz * sr;

    // atan2 on the tilt-corrected components gives the true horizontal heading
    float heading = atan2f(-myH, mxH) * RAD_TO_DEG;
```

```
    // Normalize to 0-360
    if (heading < 0.0f) heading += 360.0f;
    if (heading >= 360.0f) heading -= 360.0f;

    return heading;
}

// ----------------- LCD DRAWING -----------------
// These helpers keep display writes grouped and cached so the shared SPI bus
// can be returned to sensors as quickly as possible.
void drawTopBar(const char *title, bool showStart, bool showMark) {
    const char* startLabel = currentStartButtonLabel();
    bool sameBar =
        !topBarDirty &&
        strcmp(topBarTitleCache, title) == 0 &&
        topBarShowStartCache == showStart &&
        topBarShowMarkCache == showMark &&
        topBarSelectionCache == toolbarSelection &&
        strcmp(topBarStartLabelCache, startLabel) == 0;
    if (sameBar) return;
    lcdBeginAccess();
    tft.fillRect(0, 0, 320, 24, TILE_ORANGE);
    tft.setTextSize(1);
    tft.setTextColor(ILI9341_WHITE, TILE_ORANGE);
    tft.setCursor(4, 8);
    tft.print(title);
    if (showStart) {
        uint16_t fill = (toolbarSelection == TOOLBAR_START) ?
ILI9341_YELLOW : ILI9341_NAVY;
        uint16_t text = (toolbarSelection == TOOLBAR_START) ? ILI9341_BLACK :
ILI9341_WHITE;
        tft.fillRoundRect(BTN_START_X, BTN_Y, BTN_START_W, BTN_H, 4, fill);
        int labelX = BTN_START_X + ((BTN_START_W - ((int)strlen(startLabel) *
6)) / 2);
        if (labelX < BTN_START_X + 8) labelX = BTN_START_X + 8;
        tft.setCursor(labelX, 9);
        tft.setTextColor(text, fill);
        tft.print(startLabel);
        tft.setTextColor(ILI9341_WHITE, TILE_ORANGE);
    }
    if (showMark) {
        uint16_t fill = (toolbarSelection == TOOLBAR_MARK) ? ILI9341_YELLOW :
ILI9341_MAROON;
        uint16_t text = (toolbarSelection == TOOLBAR_MARK) ? ILI9341_BLACK :
ILI9341_WHITE;
        tft.fillRoundRect(BTN_MARK_X, BTN_Y, BTN_MARK_W, BTN_H, 4, fill);
```

```
        tft.setCursor(BTN_MARK_X + 10, 9);
        tft.setTextColor(text, fill);
        tft.print("MARK");
        tft.setTextColor(ILI9341_WHITE, TILE_ORANGE);
    }
    uint16_t closeFill = (toolbarSelection == TOOLBAR_CLOSE) ?
ILI9341_YELLOW : ILI9341_RED;
    uint16_t closeText = (toolbarSelection == TOOLBAR_CLOSE) ?
ILI9341_BLACK : ILI9341_WHITE;
    tft.fillRoundRect(BTN_CLOSE_X, BTN_Y, BTN_CLOSE_W, BTN_H, 4,
closeFill);
    tft.setCursor(BTN_CLOSE_X + 14, 9);
    tft.setTextColor(closeText, closeFill);
    tft.print("X");
    strncpy(topBarTitleCache, title, sizeof(topBarTitleCache) - 1);
    topBarTitleCache[sizeof(topBarTitleCache) - 1] = '\0';
    topBarShowStartCache = showStart;
    topBarShowMarkCache = showMark;
    topBarSelectionCache = toolbarSelection;
    strncpy(topBarStartLabelCache, startLabel, sizeof(topBarStartLabelCache) -
1);
    topBarStartLabelCache[sizeof(topBarStartLabelCache) - 1] = '\0';
    topBarDirty = false;
    lcdEndAccess();
}

bool hitTile(int x, int y, const TileCfg& t) {
    return x >= t.x && x <= (t.x + t.w) &&
           y >= t.y && y <= (t.y + t.h);
}

// Draws the BLE connection tile with a label that reflects current BLE state.
// Called from drawMenu() and serviceBLE() for incremental redraws.
void drawBleTile() {
    const char* label;
    if (bleState == BLE_ADVERTISING) {
        label = "CONNECTING...";
    } else if (bleState == BLE_CONNECTED) {
        label = "DISCONNECT FROM APP";
    } else {
        label = "CONNECT TO APP";
    }
    bool selected = (menuSelection == 4);
    uint16_t border    = selected ? ILI9341_YELLOW : ILI9341_WHITE;
    uint16_t textColor = selected ? ILI9341_YELLOW : ILI9341_WHITE;
```

```
    uint16_t fill      = (bleState == BLE_CONNECTED) ? (uint16_t)0x0760  // dark
teal when connected
                                                : (uint16_t)0x2945; // dark blue-grey otherwise
    tft.fillRoundRect(BLE_TILE_X, BLE_TILE_Y, BLE_TILE_W, BLE_TILE_H, 10,
fill);
    tft.drawRoundRect(BLE_TILE_X, BLE_TILE_Y, BLE_TILE_W, BLE_TILE_H,
10, border);
    if (selected) tft.drawRoundRect(BLE_TILE_X+2, BLE_TILE_Y+2,
BLE_TILE_W-4, BLE_TILE_H-4, 8, border);
    // Centre text horizontally; textSize=2 → 12px per char, 16px tall
    int textW  = (int)strlen(label) * 12;
    int labelX = BLE_TILE_X + (BLE_TILE_W - textW) / 2;
    int labelY = BLE_TILE_Y + (BLE_TILE_H - 16) / 2;
    tft.setTextSize(2);
    tft.setTextColor(textColor);
    tft.setCursor(labelX, labelY);
    tft.print(label);
}

void drawTile(const TileCfg& t) {
    bool selected = false;
    if (&t == &TILE_THERM && menuSelection == 0) selected = true;
    if (&t == &TILE_USS && menuSelection == 1) selected = true;
    if (&t == &TILE_INS && menuSelection == 2) selected = true;
    if (&t == &TILE_FUS && menuSelection == 3) selected = true;
    uint16_t border = selected ? ILI9341_YELLOW : t.border;
    uint16_t textColor = t.selectable ? (selected ? ILI9341_YELLOW :
ILI9341_WHITE) : ILI9341_DARKGREY;
    if (t.fill != ILI9341_BLACK) tft.fillRoundRect(t.x, t.y, t.w, t.h, 12, t.fill);
    tft.drawRoundRect(t.x, t.y, t.w, t.h, 12, border);
    if (selected) tft.drawRoundRect(t.x + 2, t.y + 2, t.w - 4, t.h - 4, 10, border);
    tft.setTextSize(2);
    tft.setTextColor(textColor);
    tft.setCursor(t.labelX, t.labelY);
    tft.print(t.label);
}

// The menu is largely static, so it is painted once per entry and then only
// updated when selection changes.
void drawMenu() {
    lcdBeginAccess();
    // Fully clear the menu canvas before redrawing tiles so remnants from
    // THERMAL / SCAN / INS screens cannot bleed through in uncovered
regions.
    tft.fillScreen(ILI9341_BLACK);
    topBarDirty = true;
```

```
    drawTile(TILE_THERM);
    drawTile(TILE_USS);
    drawTile(TILE_INS);
    drawTile(TILE_FUS);
    drawBleTile();
    menuDrawn = true;
    lcdEndAccess();
}
void drawInsBase() {
    lcdBeginAccess();
    tft.fillScreen(ILI9341_BLACK);
    topBarDirty = true;
    tft.drawFastHLine(0, 27, 320, ILI9341_DARKGREY);
    tft.drawFastVLine(0, 27, 112, ILI9341_DARKGREY);
    tft.drawFastVLine(319, 27, 112, ILI9341_DARKGREY);
    tft.drawRect(0, 148, 320, 92, ILI9341_DARKGREY);
    lcdEndAccess();
    drawTopBar("INS MAP", true, true);
}
// Handles touchscreen taps.
// Menu taps activate tiles; top-bar taps close screens or activate START/MARK.

void handleTouchXY(int x, int y) {
    if (currentMode == MENU) {
        if (hitTile(x, y, TILE_THERM)) { moveMenuSelection(0); currentMode =
THERMAL_LOADING; return; }
        if (hitTile(x, y, TILE_USS)) { moveMenuSelection(1); currentMode =
PROXIMITY; return; }
        if (hitTile(x, y, TILE_INS)) { moveMenuSelection(2); currentMode =
INS_MODE; return; }
        if (hitTile(x, y, TILE_FUS)) { moveMenuSelection(3); currentMode =
SCAN_LOADING; return; }
        if (x >= BLE_TILE_X && x <= (BLE_TILE_X + BLE_TILE_W) &&
            y >= BLE_TILE_Y && y <= (BLE_TILE_Y + BLE_TILE_H)) {
            moveMenuSelection(4);
            toggleBleConnection();
        }
        return;
    }
    if ((currentMode == THERMAL ||
         currentMode == THERMAL_LOADING) && y < 24) {
        toolbarSelection = TOOLBAR_CLOSE;
        stopAudioNow(); resetAudioState();
        ensureThermalActive(false); thermalHumanDetected=false;
presenceScore=0;
        currentMode = MENU;
```

```
        return;
    }
    // if (currentMode == SCAN_LOADING && y < 24) {
    //     toolbarSelection = TOOLBAR_CLOSE;
    //     stopAudioNow(); resetAudioState();
    //     ensureThermalActive(false); thermalHumanDetected=false;
presenceScore=0;
    //     currentMode = MENU;
    //     return;
    // }
    if (x >= BTN_CLOSE_X && x <= (BTN_CLOSE_X + BTN_CLOSE_W) &&
          y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
        toolbarSelection = TOOLBAR_CLOSE;
        topBarDirty = true;
        activateToolbarSelection();
        return;
    }
            if (currentMode == INS_MODE) {
                if (x >= BTN_START_X && x <= (BTN_START_X + BTN_START_W)
&&
                        y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
            toolbarSelection = TOOLBAR_START;
            topBarDirty = true;
            drawTopBar("INS MAP", true, true);
            activateToolbarSelection();
            return;
        }
        if (x >= BTN_MARK_X && x <= (BTN_MARK_X + BTN_MARK_W) &&
              y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
            toolbarSelection = TOOLBAR_MARK;
            topBarDirty = true;
            drawTopBar("INS MAP", true, true);
            activateToolbarSelection();
            return;
                }
              }
              if (currentMode == PROXIMITY) {
                if (x >= BTN_START_X && x <= (BTN_START_X + BTN_START_W)
&&
                        y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
                    toolbarSelection = TOOLBAR_START;
                    topBarDirty = true;
                    drawTopBar("RADAR", true, false);
                    activateToolbarSelection();
                    return;
                }
```

```
            }
            if (currentMode == SCAN_MODE) {
                if (x >= BTN_START_X && x <= (BTN_START_X + BTN_START_W)
&&
                        y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
                    toolbarSelection = TOOLBAR_START;
                    topBarDirty = true;
                    drawTopBar("SCAN", true, scanModeShowsMarkButton());
                    activateToolbarSelection();
                    return;
                }
                if (scanModeShowsMarkButton() &&
                    x >= BTN_MARK_X && x <= (BTN_MARK_X + BTN_MARK_W)
&&
                    y >= BTN_Y && y <= (BTN_Y + BTN_H)) {
                    toolbarSelection = TOOLBAR_MARK;
                    topBarDirty = true;
                    drawTopBar("SCAN", true, true);
                    activateToolbarSelection();
                    return;
                }
            }
}
// ------------------------THERMAL DETECTION-----------------------------
// Sample the thermal sensor more slowly than the main loop, smooth the raw
// frame, estimate ambient, and then classify it with the shared detector.
void updateThermalDetection() {
  // Throttle to ~10 fps — no need to run faster than the MLX90640 refresh rate
  static uint32_t lastSampleMs = 0;
  if (millis() - lastSampleMs < 100) return;
  lastSampleMs = millis();

  // Bail early if hardware isn't ready or display buffers weren't allocated
  if (!mlxReady || !upscaled || !prevThermal) {
    thermalFrameValid = false;
    thermalHumanDetected = false;
    return;
  }

  // Read raw frame data from MLX90640 over I2C — retry up to 6 times on
failure
  uint16_t frameData[1024];
  bool ok = false;
  for (int retry = 0; retry < 6; retry++) {
    int status = MLX90640_GetFrameData(MLX90640_ADDR, frameData);
    if (status >= 0) {
```

```
      float Ta = MLX90640_GetTa(frameData, &mlx90640);   // Compute ambient
temperature from the frame header bytes
      // Convert raw ADC values to temperature array using stored calibration
params
      // TA_SHIFT subtracts a fixed offset to compensate for self-heating of the
module
      MLX90640_CalculateTo(frameData, &mlx90640, EMISSIVITY, Ta -
TA_SHIFT, mlxRaw);
      ok = true;
      break;
    }
    delay(5);
  }
  if (!ok) {
    thermalFrameValid = false;
    thermalHumanDetected = false;
    return;
  }

  // Apply exponential moving average to smooth frame-to-frame noise
  // alpha=0.30 means each new frame contributes 30% to the filtered value
  const float alpha = 0.30f;
  if (!mlxInitDone) {
    for (int i = 0; i < 768; i++) mlxFilt[i] = mlxRaw[i];  // On first frame, seed the filter
with raw values directly so there's no lag
    mlxInitDone = true;
    thermalWarmupFrames = 0;
  } else {
    for (int i = 0; i < 768; i++) mlxFilt[i] += alpha * (mlxRaw[i] - mlxFilt[i]); // Each
pixel
  }

  // Discard the first 4 frames to avoid hot pixels
  // during initialization before its internal calibration stabilizes
  if (thermalWarmupFrames < 4) {
    thermalWarmupFrames++;
    thermalFrameValid = false;
    prevThermalValid = false;
    return;
  }

  // Compute scene statistics used for palette scaling and ambient estimation
  float sceneSum = 0.0f;
  int sceneCount = 0;
  float sceneMax = -1000.0f;
  float coolMin = 1000.0f;
```

```
  // Single pass over all pixels to find min/max/mean
  for (int i = 0; i < 768; i++) {
    float t = mlxFilt[i];
    if (!isfinite(t)) continue;
    sceneSum += t;
    sceneCount++;
    if (t > sceneMax) sceneMax = t;
    if (t < coolMin) coolMin = t;
  }
  thermalSceneAvgC = (sceneCount > 0) ? (sceneSum / sceneCount) : NAN;

  // Estimate ambient by averaging only the coolest pixels
  float coolSum = 0.0f;
  int coolCount = 0;
  float coolBandMax = coolMin + 2.0f;
  for (int i = 0; i < 768; i++) {
    float t = mlxFilt[i];
    if (!isfinite(t)) continue;
    if (t <= coolBandMax) {
      coolSum += t;
      coolCount++;
    }
  }

  // Run the shared thermal human detector - returns true if a human is present
  bool present = false;
  bool wasDetected=thermalHumanDetected;
  present = ThermalHuman::detectHumanThermal(
      thermalDetectorState,
      mlxFilt,
      millis(),
      thermalAmbientC,
      temperatureC,          // BMP temp reference
      thermalDetectorCfg,
      thermalDetectorFeatures,   // filled with blob geometry on return
      thermalDetectorScores);    // filled with confidence scores on return

  // Use cool-band average as ambient if we had enough cool pixels, otherwise
fall back to scene mean
  if (coolCount >= 16) thermalAmbientC = coolSum / coolCount;
  else thermalAmbientC = thermalSceneAvgC;

  // Pull display metrics out of the detector feature struct
  thermalMaxC = isfinite(thermalDetectorFeatures.max_target_temp) ?
      thermalDetectorFeatures.max_target_temp : sceneMax;
  thermalHotPixelCount = thermalDetectorFeatures.hot_pixel_count;
```

```
  // Scale confidence float to 0:100 integer for display
  thermalConfidencePct = constrain((int)lroundf(100.0f *
thermalDetectorScores.confidence), 0, 100);
  // Scale human presence score to 0:30 for the hysteresis counter
  presenceScore = constrain((int)lroundf(30.0f *
thermalDetectorScores.human_presence_score), 0, 30);
  thermalHumanDetected = present;
  thermalFrameValid = true;

  // Debounce alert trigger — require 2 consecutive positive frames
  // before firing to avoid single-frame false positives
  static uint8_t detConfirmCount = 0;
if (!thermalHumanDetected) {
  detConfirmCount = 0;

  // Re-arm only after the detector goes clear, so a continuous human does not
  // spam the alert. The next new detection can alert again.
  if (!alertTriggered) {
    alertArmed  = true;
    alertPlayed = false;
  }

} else if (!wasDetected) {
  detConfirmCount = 1;

} else if (detConfirmCount > 0) {
  detConfirmCount++;

  if (detConfirmCount >= 2) {
    detConfirmCount = 0;
    triggerAlert();
  }
}
}


//---------------------- THERMAL SCREEN -------------------------------------
static void redrawThermalImageRegion(
    int ox, int oy,
    int x0, int y0, int x1, int y1,
    float lo, float hi
) {
    x0 = max(0, x0);
    y0 = max(0, y0);
    x1 = min(THERM_RENDER_W - 1, x1);
    y1 = min(THERM_RENDER_H - 1, y1);
```

```
    if (x0 > x1 || y0 > y1) return;
    for (int y = y0; y <= y1; y++) {
        for (int x = x0; x <= x1; x++) {
            float t = upscaled[y * THERM_RENDER_W + x];
            if (x > 0 && x < (THERM_RENDER_W - 1) && y > 0 && y <
(THERM_RENDER_H - 1)) {
                t =
                    0.50f * t +
                    0.125f * upscaled[y * THERM_RENDER_W + (x - 1)] +
                    0.125f * upscaled[y * THERM_RENDER_W + (x + 1)] +
                    0.125f * upscaled[(y - 1) * THERM_RENDER_W + x] +
                    0.125f * upscaled[(y + 1) * THERM_RENDER_W + x];
            }
            uint16_t c = thermalColor(t, lo, hi);
            int idx = y * THERM_RENDER_W + x;
            tft.fillRect(ox + x * THERM_PIXEL_W, oy + y * THERM_PIXEL_H,
THERM_PIXEL_W, THERM_PIXEL_H, c);
            prevThermal[idx] = c;
        }
    }
}

// Limit render rate — 180ms gives ~5.5 fps which matches MLX90640 output
void runThermal() {
  static char prevHud1[40] = "";
  static char prevHud2[40] = "";
  static char prevHud3[40] = "";
  static int prevBoxX = 0;
  static int prevBoxY = 0;
  static int prevBoxW = 0;
  static int prevBoxH = 0;
  static bool prevBoxValid = false;
  static uint32_t lastRenderMs = 0;
  static bool alertOverlayDrawn = false;
  uint32_t renderInterval = 180;
  if (millis() - lastRenderMs < renderInterval) return;
  lastRenderMs = millis();

  // While the alert audio is playing, show a fullscreen overlay instead of the
thermal image
  // This avoids the thermal image rendering underneath the alert banner
  if (alertTriggered && (audioState == AUDIO_STARTING ||
                         audioState == AUDIO_PLAYING  ||
                         audioState == AUDIO_FINISHING)) {
    if (!alertOverlayDrawn) {
      lcdBeginAccess();
```

```
    tft.fillRoundRect(40, 100, 240, 40, 8, ILI9341_GREEN);
    tft.setTextSize(2);
    tft.setTextColor(ILI9341_BLACK, ILI9341_GREEN);
    tft.setCursor(52, 114);
    tft.print("HUMAN DETECTED");
    lcdEndAccess();
    alertOverlayDrawn = true;
  }
  return;
}

// Alert finished — invalidate caches so full thermal image redraws cleanly
if (alertOverlayDrawn) {
  alertOverlayDrawn = false;
  prevThermalValid = false;
  prevBoxValid = false;
  prevHud1[0] = '\0';
  prevHud2[0] = '\0';
  prevHud3[0] = '\0';
}
// Lambda to overwrite a single HUD text row — fills the full width with black first
// to erase any previous longer string before printing the new one
auto drawHudLine = [&](int y, const char* txt, uint16_t fg){
  tft.setTextSize(1);
  tft.fillRect(0, y, 320, 10, ILI9341_BLACK);
      tft.setTextColor(fg, ILI9341_BLACK);
      tft.setCursor(2, y);
      tft.print(txt);
  };

  // On first entry after a screen transition, invalidate all HUD caches
  // so every line gets redrawn even if the content didn't change
  if (!thermalHudInit) {
      prevHud1[0] = '\0';
      prevHud2[0] = '\0';
      prevHud3[0] = '\0';
      prevBoxValid = false;
      thermalHudInit = true;
  }

  // Show error message rows if hardware is missing or buffers failed to allocate
  if (!mlxReady) {
      const char* l1 = "MLX init failed";
      const char* l2 = "Det: --";
      const char* l3 = "Class: --";
      lcdBeginAccess();
```

```
        if (strcmp(prevHud1, l1) != 0) { drawHudLine(210, l1, ILI9341_YELLOW);
strncpy(prevHud1, l1, sizeof(prevHud1) - 1); prevHud1[sizeof(prevHud1) - 1] =
'\0'; }
        if (strcmp(prevHud2, l2) != 0) { drawHudLine(222, l2, ILI9341_YELLOW);
strncpy(prevHud2, l2, sizeof(prevHud2) - 1); prevHud2[sizeof(prevHud2) - 1] =
'\0'; }
        if (strcmp(prevHud3, l3) != 0) { drawHudLine(232, l3, ILI9341_YELLOW);
strncpy(prevHud3, l3, sizeof(prevHud3) - 1); prevHud3[sizeof(prevHud3) - 1] =
'\0'; }
        lcdEndAccess();
        return;
    }
  if (!upscaled || !prevThermal) {
    const char* l1 = "Thermal RAM low";
    const char* l2 = "Det: --";
    const char* l3 = "Class: --";
    lcdBeginAccess();
    if (strcmp(prevHud1, l1) != 0) { drawHudLine(210, l1, ILI9341_YELLOW);
strncpy(prevHud1, l1, sizeof(prevHud1) - 1); prevHud1[sizeof(prevHud1) - 1] =
'\0'; }
    if (strcmp(prevHud2, l2) != 0) { drawHudLine(222, l2, ILI9341_YELLOW);
strncpy(prevHud2, l2, sizeof(prevHud2) - 1); prevHud2[sizeof(prevHud2) - 1] =
'\0'; }
    if (strcmp(prevHud3, l3) != 0) { drawHudLine(232, l3, ILI9341_YELLOW);
strncpy(prevHud3, l3, sizeof(prevHud3) - 1); prevHud3[sizeof(prevHud3) - 1] =
'\0'; }
    lcdEndAccess();
    return;
  }

  // Run the sensor pipeline — samples the MLX90640, filters, and classifies
  updateThermalDetection();
  if (!thermalFrameValid) {
    const char* l1 = mlxInitDone ? "MLX settling..." : "MLX retry...";
    if (strcmp(prevHud1, l1) != 0) {
        lcdBeginAccess();
        drawHudLine(222, l1, ILI9341_YELLOW);
      lcdEndAccess();
      strncpy(prevHud1, l1, sizeof(prevHud1) - 1);
      prevHud1[sizeof(prevHud1) - 1] = '\0';
    }
    return;
  }
  upscaleBilinear4x(mlxFilt, upscaled);  // Bilinearly upscale the 32x24 MLX grid
to 64x48 for smoother rendering
```

```
  // Find the temperature range of the filtered image for palette scaling
  float sceneMin = 1000.0f;
  float sceneMax = -1000.0f;
  bool haveSceneRange = false;
    for (int i = 0; i < 768; i++) {
        float t = mlxFilt[i];
        if (!isfinite(t)) continue;
        haveSceneRange = true;
        if (t < sceneMin) sceneMin = t;
        if (t > sceneMax) sceneMax = t;
    }

  // Set palette bounds anchored around ambient so the background stays
cool/blue
  // and human-temperature pixels remain warm/red regardless of scene
temperature
  float lo = isfinite(thermalSceneAvgC) ? (thermalSceneAvgC - 1.8f) :
(haveSceneRange ? sceneMin : 20.0f);
  float hi = isfinite(thermalSceneAvgC) ? (thermalSceneAvgC + 3.8f) :
(haveSceneRange ? sceneMax : 35.0f);
  if (isfinite(thermalMaxC)) {               // Extend hi to show the hottest detected
target clearly
    hi = max(hi, thermalMaxC + 0.2f);
  }
  if (haveSceneRange) {                      // Ensure palette covers the actual scene
range
    lo = min(lo, sceneMin - 0.2f);
    hi = min(hi, sceneMax + 0.2f);
  }
  if (!isfinite(lo) || !isfinite(hi) || (hi - lo) < 2.5f) {       // Fallback if range is
degenerate
    if (haveSceneRange) {
      lo = sceneMin - 0.2f;
      hi = max(sceneMax + 0.2f, lo + 2.5f);
    } else {
      lo = 20.0f;
      hi = 35.0f;
    }
  }

  // Slowly track palette bounds with EMA so sudden temperature changes don't
  // cause jarring color jumps — alpha=0.12 gives ~8 frame smoothing at 5.5fps
  static float paletteLo = NAN;
  static float paletteHi = NAN;
  if (!isfinite(paletteLo) || !isfinite(paletteHi) || !prevThermalValid) {
    paletteLo = lo; paletteHi = hi;   // seed on first frame
```

```
  } else {
    paletteLo += 0.12f * (lo - paletteLo);
    paletteHi += 0.12f * (hi - paletteHi);
  }
  lo = paletteLo; hi = paletteHi;

    // Take the SPI bus for the full render pass
    lcdBeginAccess();
    const int ox = 32, oy = 24;      // pixel offset to center the 256x192 image on the
320x240 screen

    // Render each upscaled pixel as a 4x4 block — apply a 5-tap smoothing
kernel
    // to adjacent pixels before colorizing to reduce blocky artifacts
    for (int y = 0; y < THERM_RENDER_H; y++) {
        for (int x = 0; x < THERM_RENDER_W; x++) {
            float t = upscaled[y * THERM_RENDER_W + x];
            // 5-tap cross kernel: 50% center + 12.5% each cardinal neighbor
            if (x > 0 && x < (THERM_RENDER_W - 1) && y > 0 && y <
(THERM_RENDER_H - 1)) {
                t =
                    0.50f * t +
                    0.125f * upscaled[y * THERM_RENDER_W + (x - 1)] +
                    0.125f * upscaled[y * THERM_RENDER_W + (x + 1)] +
                    0.125f * upscaled[(y - 1) * THERM_RENDER_W + x] +
                    0.125f * upscaled[(y + 1) * THERM_RENDER_W + x];
            }
            // Map temperature to a blue-green-yellow-red color using the palette
bounds
            uint16_t c = thermalColor(t, lo, hi);
            int idx = y * THERM_RENDER_W + x;
            if (!prevThermalValid || prevThermal[idx] != c) {        // Only write pixels
that changed since the last frame to reduce SPI traffic
                tft.fillRect(ox + x * THERM_PIXEL_W, oy + y * THERM_PIXEL_H,
THERM_PIXEL_W, THERM_PIXEL_H, c);
                prevThermal[idx] = c;
            }
        }
    }
    if (prevBoxValid) {
        int px0 = prevBoxX / THERM_PIXEL_W - 1;
        int py0 = prevBoxY / THERM_PIXEL_H - 1;
        int px1 = (prevBoxX + prevBoxW) / THERM_PIXEL_W + 1;
        int py1 = (prevBoxY + prevBoxH) / THERM_PIXEL_H + 1;
        redrawThermalImageRegion(ox, oy, px0, py0, px1, py1, lo, hi);
        prevBoxValid = false;
```

```
    }
    // Each MLX sensor pixel maps to (THERM_RENDER_W/kWidth *
THERM_PIXEL_W) screen pixels.
    // kWidth=32, THERM_RENDER_W=64 → upscale=2; THERM_PIXEL_W=4
→ 2*4=8 screen px per MLX px.
    // Draw the bounding box AFTER all pixels are rendered so fillRect calls don't
overwrite it.
    // Show bounding box for any blob the classifier considers a candidate (not just
full confirm).
    // Box color: solid white border = confirmed human; dotted/dim = investigating.
    bool bboxCandidate = (thermalDetectorScores.best_class !=
ThermalHuman::NO_HUMAN)
                         && thermalDetectorFeatures.hot_pixel_count > 0
                         && thermalDetectorFeatures.bbox_w > 0
                         && thermalDetectorFeatures.bbox_h > 0;
    if (bboxCandidate) {
        const int scaleX = (THERM_RENDER_W / ThermalHuman::kWidth) *
THERM_PIXEL_W; // 8
        const int scaleY = (THERM_RENDER_H / ThermalHuman::kHeight) *
THERM_PIXEL_H; // 8
        int bx = ox + thermalDetectorFeatures.bbox_x * scaleX;
        int by = oy + thermalDetectorFeatures.bbox_y * scaleY;
        int bw = thermalDetectorFeatures.bbox_w * scaleX;
        int bh = thermalDetectorFeatures.bbox_h * scaleY;
        uint16_t boxColor = thermalHumanDetected ? ILI9341_WHITE :
ILI9341_DARKGREY;
        tft.drawRect(bx,     by,     bw,     bh,     ILI9341_BLACK);
        tft.drawRect(bx + 1, by + 1, bw - 2, bh - 2, boxColor);
        prevBoxX = bx;
        prevBoxY = by;
        prevBoxW = bw;
        prevBoxH = bh;
        prevBoxValid = true;
    }
    prevThermalValid = true;
    thermalImagePrimed = true;

  // Build HUD text strings — only redraw rows whose content changed
  char line1[40], line2[40], line3[40];
  const char* detLabel = thermalHumanDetected ? "YES" :
     (thermalDetectorScores.best_class != ThermalHuman::NO_HUMAN ?
"INVST" : "NO");
  snprintf(line1, sizeof(line1), "Max:%.1fF Amb:%.1fF", cToF(thermalMaxC),
cToF(thermalAmbientC));
  snprintf(line2, sizeof(line2), "Det:%s P:%d C:%d", detLabel, presenceScore,
thermalConfidencePct);
```

```
  // Line 3 content and color depend on mode. In SCAN_MODE the review screen
  // replaces the standard "class / mean temp / hot pixel" debug info with a
  // live Final %/Decision readout driven by the fusion engine. runScan()
  // re-pushes the current thermal confidence into the scan runtime and
  // re-runs fuseResults() every frame while the thermal screen is visible,
  // so this line climbs right alongside the C: value in line 2 the longer
  // the subject stays in the MLX frame.
  uint16_t line3Color = ILI9341_CYAN;
  if (currentMode == SCAN_MODE) {
    const int finalPct = constrain((int)lroundf(100.0f * scanDisplayConfidence()), 0,
100);
    const char* decision = (finalPct >= 70) ? "Human"
                         : (finalPct >= 40) ? "Possible"
                         :                    "No human";
    snprintf(line3, sizeof(line3), "Final:%d%% %s", finalPct, decision);
    line3Color = (finalPct >= 70) ? ILI9341_GREEN
               : (finalPct >= 40) ? ILI9341_YELLOW
               :                    ILI9341_DARKGREY;
  } else {
    snprintf(line3, sizeof(line3), "%s M:%.1fF A:%d",
             ThermalHuman::className(thermalDetectorScores.best_class),
             cToF(thermalDetectorFeatures.mean_target_temp),
             thermalHotPixelCount);
  }
  // Check each HUD line before writing — avoids SPI traffic for unchanged rows
  if (strcmp(prevHud1, line1) != 0) {
       drawHudLine(210, line1, ILI9341_WHITE);
       strncpy(prevHud1, line1, sizeof(prevHud1) - 1); prevHud1[sizeof(prevHud1)
- 1] = '\0';
    }
  if (strcmp(prevHud2, line2) != 0) {
    drawHudLine(222, line2, thermalHumanDetected ? ILI9341_GREEN :
ILI9341_YELLOW);
    strncpy(prevHud2, line2, sizeof(prevHud2) - 1); prevHud2[sizeof(prevHud2) -
1] = '\0';
  }
  if (strcmp(prevHud3, line3) != 0) {
    drawHudLine(232, line3, line3Color);
    strncpy(prevHud3, line3, sizeof(prevHud3) - 1); prevHud3[sizeof(prevHud3) -
1] = '\0';
  }
  // BLE thermal HUD stream — sent at 2 Hz while connected in THERMAL mode
  static uint32_t lastBleHudMs = 0;
  if (bleState == BLE_CONNECTED && millis() - lastBleHudMs >= 500) {
    lastBleHudMs = millis();
    char bleMsg[96];
```

```
    snprintf(bleMsg, sizeof(bleMsg),
"TH|%s|%d|%d|%.1f|%.1f|%.1f|%d|%d|%d|%d",
      thermalHumanDetected ? "YES" : "NO",
      presenceScore,
      thermalConfidencePct,
      thermalMaxC,
      thermalAmbientC,
      thermalSceneAvgC,
      thermalDetectorFeatures.bbox_x,
      thermalDetectorFeatures.bbox_y,
      thermalDetectorFeatures.bbox_w,
      thermalDetectorFeatures.bbox_h);
    bleSendLog(bleMsg);
  }
  lcdEndAccess();
}

// -------------------- LOADING SCREEN THERMAL/SCAN -----------------
static void drawThermalBase() {
  lcdBeginAccess();
  tft.fillScreen(ILI9341_BLACK);
  lcdEndAccess();
  topBarDirty = true;
  drawTopBar("THERMAL", false, false);
  prevThermalValid = false;
  mlxInitDone = false;
  thermalHudInit = false;
}

static void drawThermalLoadingScreen() {
  lcdBeginAccess();
  tft.fillScreen(ILI9341_BLACK);
  tft.fillRect(0,0,320,24,TILE_ORANGE);
  tft.setTextSize(1);
  tft.setTextColor(ILI9341_WHITE,TILE_ORANGE);
  tft.setCursor(4,8);
  tft.print("THERMAL");

tft.fillRoundRect(BTN_CLOSE_X,BTN_Y,BTN_CLOSE_W,BTN_H,4,ILI9341_RE
D);
  tft.setCursor(BTN_CLOSE_X+14,9);
  tft.setTextColor(ILI9341_WHITE,ILI9341_RED);
  tft.print("X");
  tft.drawCircle(160,110,52,ILI9341_DARKGREY);
  tft.drawCircle(160,110,51,TILE_ORANGE);
  tft.fillCircle(160,110,30,tft.color565(60,0,0));
```

```
  tft.fillCircle(160,110,20,tft.color565(140,30,0));
  tft.fillCircle(160,110,12,tft.color565(220,80,0));
  tft.fillCircle(160,110, 5,ILI9341_YELLOW);
  tft.setTextSize(1);
  tft.setTextColor(ILI9341_WHITE);
  tft.setCursor(98,175);
  tft.print("Initializing sensor...");
  tft.setTextColor(ILI9341_DARKGREY);
  tft.setCursor(82,220);
  tft.print("Hold tight, warming up...");
  lcdEndAccess();
}

static void drawScanLoadingScreen() {
  lcdBeginAccess();
  tft.fillScreen(ILI9341_BLACK);
  tft.fillRect(0,0,320,24,TILE_ORANGE);
  tft.setTextSize(1);
  tft.setTextColor(ILI9341_WHITE,TILE_ORANGE);
  tft.setCursor(4,8);
  tft.print("SCAN");

tft.fillRoundRect(BTN_CLOSE_X,BTN_Y,BTN_CLOSE_W,BTN_H,4,ILI9341_RE
D);
  tft.setCursor(BTN_CLOSE_X+14,9);
  tft.setTextColor(ILI9341_WHITE,ILI9341_RED);
  tft.print("X");
  tft.drawCircle(160,110,52,ILI9341_DARKGREY);
  tft.drawCircle(160,110,51,TILE_ORANGE);
  tft.fillCircle(160,110,30,tft.color565(60,0,0));
  tft.fillCircle(160,110,20,tft.color565(140,30,0));
  tft.fillCircle(160,110,12,tft.color565(220,80,0));
  tft.fillCircle(160,110, 5,ILI9341_YELLOW);
  tft.setTextSize(1);
  tft.setTextColor(ILI9341_WHITE);
  tft.setCursor(112,175);
  tft.print("Initializing scan...");
  tft.setTextColor(ILI9341_DARKGREY);
  tft.setCursor(88,220);
  tft.print("Hold tight, starting up...");
  lcdEndAccess();
}

void drawLoadingScreen(){
  drawThermalLoadingScreen();
}
```

```
void updateLoadingBar(float progress){
  static float lastProg=-1.0f;
  if(fabsf(progress-lastProg)<0.01f)return;
  lastProg=progress;
  lcdBeginAccess();
  const int barX=40,barY=192,barW=240,barH=8;
  tft.drawRoundRect(barX,barY,barW,barH,4,ILI9341_DARKGREY);
  int filled=(int)(progress*(float)(barW-4));
  if(filled>0)tft.fillRoundRect(barX+2,barY+2,filled,barH-4,2,TILE_ORANGE);
  lcdEndAccess();
  }

// ----------------------- PROXIMITY SCREEN ----------------------------
void runProximity() {
    static uint32_t lastMs = 0;
    static bool prevTooClose = false;
    static bool prevRadarOn = false;
    static bool sonarBaseDrawn = false;
    static float trailYaw[6];  // [0]=newest … [5]=oldest; -9999=invalid
    static float lastHumanRadarM = NAN;
    static int16_t lastDotPx = 0;
    static int16_t lastDotPy = 0;
    static bool lastDotValid = false;
    static bool sonarBufInit = false;
    static char prevL1[40] = "";
    static char prevL2[40] = "";
    static char prevL3[40] = "";
    static char prevL4[40] = "";
    static bool radarKilledByProx = false;
    // Scan progress bar caches — track the last-drawn fill and color so the
    // bar only repaints when something actually changed. -1 forces the first
    // draw on entry and after any full-screen invalidation below (too-close
    // banner, screen re-init, etc).
    static int      prevScanBarFill  = -1;
    static uint16_t prevScanBarColor = 0;
    static bool     scanBarBorderDrawn = false;

    if (millis() - lastMs < 120) return;
    lastMs = millis();

    if (!proxScreenInit) {
        lcdBeginAccess();
        tft.fillScreen(ILI9341_BLACK);
        tft.drawRect(0, 24, 320, 166, ILI9341_DARKGREY);
        lcdEndAccess();
```

```
        topBarDirty = true;
        drawTopBar("RADAR", true, false);
        sonarBaseDrawn = false;
        for (int i = 0; i < 6; i++) trailYaw[i] = -9999.0f;
        lastHumanRadarM = NAN;
        lastDotValid = false;
        sonarBufInit = true;
        proxScreenInit = true;
        // If proxRadarRestartPrompt is set (flag just cleared), keep
prevTooClose=true
        // so the "Area clear" stripe draws on the first render after reinit.
        prevTooClose = proxRadarRestartPrompt || proxTooCloseLatched;
        prevRadarOn = false;
        prevL1[0] = prevL2[0] = prevL3[0] = prevL4[0] = '\0';
        prevScanBarFill    = -1;
        scanBarBorderDrawn = false;
    }

updateProximityReading();

bool tooClose = proxTooCloseLatched;

if (tooClose && !radarKilledByProx) {
    radarKillHard();
    radarKilledByProx = true;
    proxRadarRestartPrompt = false;
}
else if (!tooClose && radarKilledByProx) {
    radarKilledByProx = false;
    proxRadarRestartPrompt = true;   // user press start
    proxRadarScopeBaseDrawn = false;
    prevL1[0] = prevL2[0] = prevL3[0] = prevL4[0] = '\0';
}

// Alert audio plays in a tight-pump in loop(). Once done, screen reinits.
// While still too close, show the persistent bottom banner and pause rendering.
if (tooClose) {
    if (!prevTooClose) {
        lcdBeginAccess();
        tft.fillRect(0, 190, 320, 50, ILI9341_RED);
        tft.setTextSize(1);
        tft.setTextColor(ILI9341_WHITE, ILI9341_RED);
        tft.setCursor(110, 208);
        tft.print("TOO CLOSE");
        lcdEndAccess();
        prevTooClose = true;
```

```
    }
    return;
}

// We only get here after distance is clear AND the final warning clip finished.
if (prevTooClose) {
    lcdBeginAccess();
    tft.fillRect(0, 190, 320, 50, ILI9341_BLACK);

    if (proxRadarRestartPrompt) {
        tft.setTextSize(1);
        tft.setTextColor(ILI9341_CYAN, ILI9341_BLACK);
        tft.setCursor(10, 207);
        tft.print("Area clear - press START");
    }

    lcdEndAccess();

    prevTooClose = false;
    prevL1[0] = prevL2[0] = prevL3[0] = prevL4[0] = '\0';
}

radarPoll();

    // If radar currently tracks a huuman, keep the lED lit
    // triggerLED() restarts the 5 sec window each pass so the LED stays on
continuosuly for as long as a target is present
    if (radarHuman) triggerLED();
    auto drawPolarMap = [&]() {
        const int CX      = 160;
        const int CY      = 110;
        const int MAP_R   = 60;    // pixel radius = 3 m
        const int LABEL_R = 68;    // spoke-label offset from centre
        const int graphX  = 2;
        const int graphY  = 26;
        const int graphW  = 316;
        const int graphH  = 162;
        uint32_t now = millis();
        // Phosphor decay colours: index 0 = current (bright), 5 = oldest (dim)
        const uint16_t sweepColors[6] = {
            0x07E0, 0x0640, 0x04A0, 0x0320, 0x01A0, 0x00A0
        };
        // Spoke labels and their pixel widths at textSize 1 (6 px/char)
        const char* const spokeLabels[8] = {
            "0","45","90","135","180","225","270","315"
        };
```

```
        const int8_t spokeLabelW[8] = { 6, 12, 12, 18, 18, 18, 18, 18 };
        // One-time buffer zero-init
        if (!sonarBufInit) {
            for (int i = 0; i < 6; i++) trailYaw[i] = -9999.0f;
            sonarBufInit = true;
        }
        // Helper: redraw 3 rings + 8 spokes + centre dot over whatever is there.
        // Called after every sweep-line update so the grid always shows through.
        auto restoreBase = [&]() {
            for (int i = 1; i <= 3; i++)
                tft.drawCircle(CX, CY, i * 20, ILI9341_DARKGREY);
            for (int i = 0; i < 8; i++) {
                float a = i * 45.0f * PI / 180.0f;
                tft.drawLine(CX, CY,
                    CX + (int)(MAP_R * sinf(a)),
                    CY - (int)(MAP_R * cosf(a)),
                    ILI9341_DARKGREY);
            }
            tft.fillCircle(CX, CY, 2, ILI9341_DARKGREY);
        };
        // Draw static base on first entry (or after mode re-init)
        if (!sonarBaseDrawn) {
            tft.fillRect(graphX, graphY, graphW, graphH, ILI9341_BLACK);
            tft.drawRect(0, 24, 320, 166, ILI9341_DARKGREY);
            restoreBase();
            // Range labels — safe from sweep erasure (sweep stays within MAP_R)
            tft.setTextSize(1);
            tft.setTextColor(ILI9341_DARKGREY, ILI9341_BLACK);
            tft.setCursor(CX + 3, CY - 18);  tft.print("1m");
            tft.setCursor(CX + 3, CY - 38);  tft.print("2m");
            tft.setCursor(CX + 3, CY - 58);  tft.print("3m");
            // Spoke-terminus degree labels (drawn once; outside MAP_R so never
erased)
            for (int i = 0; i < 8; i++) {
                float a  = i * 45.0f * PI / 180.0f;
                int   lx = CX + (int)(LABEL_R * sinf(a)) - spokeLabelW[i] / 2;
                int   ly = CY - (int)(LABEL_R * cosf(a)) - 4;
                tft.setCursor(lx, ly);
                tft.print(spokeLabels[i]);
            }
            sonarBaseDrawn = true;
        }
        // --- Decay-trail update ---
        // 1. Erase the line that is about to fall off the end of the buffer
        if (trailYaw[5] > -9000.0f) {
            float pr = trailYaw[5] * PI / 180.0f;
```

```
        tft.drawLine(CX, CY,
            CX + (int)(MAP_R * sinf(pr)),
            CY - (int)(MAP_R * cosf(pr)),
            ILI9341_BLACK);
    }
    // 2. Shift buffer and push current heading
    for (int i = 5; i > 0; i--) trailYaw[i] = trailYaw[i - 1];
    trailYaw[0] = yawDeg;
    // 3. Repaint all trail segments darkest-first so newer lines paint over older
    for (int i = 5; i >= 0; i--) {
        if (trailYaw[i] < -9000.0f) continue;
        float tr = trailYaw[i] * PI / 180.0f;
        tft.drawLine(CX, CY,
            CX + (int)(MAP_R * sinf(tr)),
            CY - (int)(MAP_R * cosf(tr)),
            sweepColors[i]);
    }
   // 4. Restore rings and spokes so the grid is always visible
    restoreBase();
    // --- Detection dot (single, most-recent hit only) ---
    // Erase the previous dot
    if (lastDotValid) {
        tft.fillCircle(lastDotPx, lastDotPy, 3, ILI9341_BLACK);
        lastDotValid = false;
    }
    // Draw a new dot if radar currently sees a human
    float radarDotM = radarCurrentDistanceM();
    float yr = trailYaw[0] * PI / 180.0f;
    if (radarHuman && isfinite(radarDotM) && radarDotM > 0.05f) {
        float r = (radarDotM / 3.0f) * MAP_R;
        if (r > MAP_R) r = MAP_R;
        lastDotPx    = (int16_t)(CX + r * sinf(yr));
        lastDotPy    = (int16_t)(CY - r * cosf(yr));
        lastDotValid = true;
        tft.fillCircle(lastDotPx, lastDotPy, 3, ILI9341_RED);
    }
    // Heading readout, top-left corner
    tft.setTextSize(1);
    tft.setTextColor(ILI9341_YELLOW, ILI9341_BLACK);
    char hdgBuf[10];
    snprintf(hdgBuf, sizeof(hdgBuf), "%3.0f\xB0 ", yawDeg);
    tft.setCursor(graphX + 2, graphY + 2);
    tft.print(hdgBuf);
};

lcdBeginAccess();
```

```
    drawPolarMap();
    auto drawLine = [&](int y, const char* txt, uint16_t fg, uint16_t bg) {
        tft.fillRect(10, y, 300, 8, bg);
        tft.setTextColor(fg, bg);
        tft.setCursor(10, y);
        tft.print(txt);
    };

    if (tooClose != prevTooClose) {
        if (tooClose) {
            tft.fillRect(0, 190, 320, 50, ILI9341_RED);
            tft.setTextSize(1);
            tft.setTextColor(ILI9341_WHITE, ILI9341_RED);
            tft.setCursor(110, 208);
            tft.print("TOO CLOSE");
        } else {
            tft.fillRect(0, 190, 320, 50, ILI9341_BLACK);
            prevL1[0] = prevL2[0] = prevL3[0] = prevL4[0] = '\0';
            prevScanBarFill    = -1;
            scanBarBorderDrawn = false;
        }
        prevTooClose = tooClose;
    }
    if (!tooClose && !proxRadarRestartPrompt) {
        tft.setTextSize(1);
        char distTxt[16];
        char brTxt[16];
        char hrTxt[16];
        char rdistTxt[24];
        char rdistFeetTxt[16];

        if (proxFiltered > 0.0f) snprintf(distTxt, sizeof(distTxt), "%.1f", proxFiltered);
            else snprintf(distTxt, sizeof(distTxt), "---");
        // Feed the display EMAs into fusion — same smoothing the user asked
        // for on the scan screen, but done via a live EMA instead of a
        // phase-1 running average (this screen has no phase boundary).
        float _fbr = fusedBr(isfinite(radarBreathBpmDisplay) ?
radarBreathBpmDisplay : radarBreathBpm);
        float _fhr = fusedHr(isfinite(radarHeartBpmDisplay)  ?
radarHeartBpmDisplay  : radarHeartBpm);
        if (isfinite(_fbr)) snprintf(brTxt, sizeof(brTxt), "%.1f", _fbr);
            else snprintf(brTxt, sizeof(brTxt), "--");
        if (isfinite(_fhr)) snprintf(hrTxt, sizeof(hrTxt), "%.1f", _fhr);
            else snprintf(hrTxt, sizeof(hrTxt), "--");

        float _tiDist = radarCurrentDistanceM();
```

```
        float displayRadarDistM = fusedDist(_tiDist);
        if (isfinite(displayRadarDistM)) {
            formatFeetInches(displayRadarDistM, rdistFeetTxt, sizeof(rdistFeetTxt));
            snprintf(rdistTxt, sizeof(rdistTxt), "%s (%.2fm)", rdistFeetTxt,
displayRadarDistM);
        } else {
            snprintf(rdistTxt, sizeof(rdistTxt), "--");
        }
        char l1[40], l2[40], l3[40], l4[40];

        snprintf(l1, sizeof(l1), "USS: %s cm  Yaw:%3.0f", distTxt, yawDeg);
        snprintf(l2, sizeof(l2), "BR: %s HR: %s", brTxt, hrTxt);
        snprintf(l3, sizeof(l3), "RDist: %s", rdistTxt);

        if (radarHuman && isfinite(displayRadarDistM)) lastHumanRadarM =
displayRadarDistM;
        // L4 doubles as the scan phase readout while the radar driver is
        // running its two-phase 30 s + 30 s sweep. The progress bar rendered
        // below visualises the same percentage — this line just gives the
        // operator the textual label + elapsed/total seconds. Once the sweep
        // is done (or idle) we fall back to the classic "State: TRACK/SCAN
        // <dist>" summary.
        bool phase1Active =
            radarScanState == RADAR_SCAN_WAIT_P1 ||
            radarScanState == RADAR_SCAN_RUNNING_P1;
        bool phase2Active =
            radarScanState == RADAR_SCAN_WAIT_P2 ||
            radarScanState == RADAR_SCAN_RUNNING_P2;
        bool scanActive = phase1Active || phase2Active;
        if (scanActive) {
            // L4 during the sweep: "Phase 1 Xs/30s" or "Phase 2 Xs/30s".
            // Elapsed only ticks while RUNNING_*; WAIT_* shows 0s waiting for
            // the radar to acquire a human.
            uint32_t elapsedMs =
                (radarScanState == RADAR_SCAN_RUNNING_P1 || radarScanState
== RADAR_SCAN_RUNNING_P2)
                    ? (millis() - radarPhaseStartMs) : 0;
            if (elapsedMs > RADAR_PHASE_MS) elapsedMs =
RADAR_PHASE_MS;
            snprintf(l4, sizeof(l4), "%s %lus/%lus",
                     phase1Active ? "Phase 1" : "Phase 2",
                     (unsigned long)(elapsedMs / 1000UL),
                     (unsigned long)(RADAR_PHASE_MS / 1000UL));
        } else if (radarScanState == RADAR_SCAN_DONE) {
            // After the sweep, the radar keeps streaming Phase 2 data so
            // radarHuman reflects the live presence. Flip between
```

```
          // "Detected" / "No detection" accordingly.
          snprintf(l4, sizeof(l4), "Complete: %s",
                radarHuman ? "Detected" : "No detection");
       } else if (isfinite(lastHumanRadarM))
          snprintf(l4, sizeof(l4), "State: %s %.2fm", radarHuman ? "TRACK" :
"SCAN", lastHumanRadarM);
       else
          snprintf(l4, sizeof(l4), "State: %s", radarHuman ? "TRACK" : "SCAN");
       if (strcmp(prevL1, l1) != 0) { drawLine(RADAR_INFO_Y + 0 *
RADAR_INFO_DY, l1, ILI9341_WHITE, ILI9341_BLACK); strcpy(prevL1, l1); }
            if (strcmp(prevL2, l2) != 0) { drawLine(RADAR_INFO_Y + 1 *
RADAR_INFO_DY, l2, ILI9341_WHITE, ILI9341_BLACK); strcpy(prevL2, l2); }
            if (strcmp(prevL3, l3) != 0 || prevRadarOn != radarEnabled) {
               drawLine(RADAR_INFO_Y + 2 * RADAR_INFO_DY, l3,
ILI9341_WHITE, ILI9341_BLACK);
               strcpy(prevL3, l3);
            }
            if (strcmp(prevL4, l4) != 0) { drawLine(RADAR_INFO_Y + 3 *
RADAR_INFO_DY, l4, TILE_ORANGE, ILI9341_BLACK); strcpy(prevL4, l4); }

            // --- Scan progress bar ---
            // Thin horizontal bar at the very bottom of the radar screen.
            // Visualises radarScanProgressPermille() so the operator can
            // see at a glance how far into Phase 1 / Phase 2 the radar
            // driver is. The bar spans both phases end-to-end: 0..50%
            // fills during Phase 1 (30 s), 50..100% during Phase 2
            // (30 s). Drained to empty when the radar is idle; painted
            // green solid when RADAR_SCAN_DONE.
            const int barX = 4;
            const int barY = 234;
            const int barW = 312;
            const int barH = 5;
            const int fillX = barX + 1;
            const int fillY = barY + 1;
            const int fillW = barW - 2;
            const int fillH = barH - 2;
            if (!scanBarBorderDrawn) {
               tft.drawRect(barX, barY, barW, barH, ILI9341_DARKGREY);
               tft.fillRect(fillX, fillY, fillW, fillH, ILI9341_BLACK);
               scanBarBorderDrawn = true;
               prevScanBarFill = -1;  // force the fill block to paint below
            }
            uint16_t perMille = radarScanProgressPermille();
            int      targetFill = (int)(((int32_t)perMille * fillW) / 1000);
            if (targetFill < 0)    targetFill = 0;
            if (targetFill > fillW) targetFill = fillW;
```

```
            uint16_t barColor = TILE_ORANGE;
            switch (radarScanState) {
                case RADAR_SCAN_RUNNING_P1:
                case RADAR_SCAN_WAIT_P1:
                    barColor = TILE_ORANGE; break;
                case RADAR_SCAN_RUNNING_P2:
                case RADAR_SCAN_WAIT_P2:
                    barColor = ILI9341_CYAN; break;
                case RADAR_SCAN_DONE:
                    barColor = ILI9341_GREEN; break;
                default:
                    barColor = ILI9341_DARKGREY; break;
            }
            if (targetFill != prevScanBarFill || barColor != prevScanBarColor) {
                tft.fillRect(fillX, fillY, fillW, fillH, ILI9341_BLACK);
                if (targetFill > 0) tft.fillRect(fillX, fillY, targetFill, fillH, barColor);
                prevScanBarFill  = targetFill;
                prevScanBarColor = barColor;
            }
        }
        else {
            if (radarEnabled) {
                tft.setTextSize(1);
                char l1[40], l2[40], l3[40];
                snprintf(l1, sizeof(l1), "USS: %.1f cm", proxFiltered);
                snprintf(l2, sizeof(l2), "Radar kept active");
                snprintf(l3, sizeof(l3), "Back up to improve view");
                if (strcmp(prevL1, l1) != 0) { drawLine(RADAR_INFO_Y + 0 *
RADAR_INFO_DY, l1, ILI9341_WHITE, ILI9341_RED); strcpy(prevL1, l1); }
                if (strcmp(prevL2, l2) != 0) { drawLine(RADAR_INFO_Y + 1 *
RADAR_INFO_DY, l2, ILI9341_WHITE, ILI9341_RED); strcpy(prevL2, l2); }
                if (strcmp(prevL3, l3) != 0) { drawLine(RADAR_INFO_Y + 2 *
RADAR_INFO_DY, l3, ILI9341_WHITE, ILI9341_RED); strcpy(prevL3, l3); }
            }
        }
        lcdEndAccess();
        prevRadarOn = radarEnabled;
    }

// The INS renderer keeps separate caches for the map, compass, and text rows
// so only changed regions are repainted on each pass.
void runINS() {
    static uint32_t lastDrawMs = 0;
    // Limit INS redraw cadence so the TFT is not hammered continuously while
    // UWB ranging and sensor updates are also active on the system.
    if (millis() - lastDrawMs < 10) return;
```

```
    lastDrawMs = millis();

    // Pause heavy LCD render while audio is active (mirrors runThermal() guard).
    static bool insAudioPaused = false;
    static bool compassBaseDrawn = false;
    static bool compassNeedleDrawn = false;
    bool blockingAudio =
    currentClip == CLIP_TOOCLOSE ||
    currentClip == CLIP_HUMAN ||
    currentClip == CLIP_SCAN;
    const bool allowBlockingAudioDuringScanReview =
        currentMode == SCAN_MODE &&
        (scanReviewSequenceActive || scanRt.state ==
ScanMode::SCAN_POSITION);

    if (blockingAudio &&
    (audioState == AUDIO_STARTING ||
     audioState == AUDIO_PLAYING ||
     audioState == AUDIO_FINISHING) &&
    !allowBlockingAudioDuringScanReview) {
        insAudioPaused = true;
        return;
    }
    if (insAudioPaused) {
        insCacheInit = false;   // force full redraw once audio finishes
        insAudioPaused = false;
    }

    if (!insScreenInit) {
        drawInsBase();
        for (int r = 0; r < MAP_H; r++) prevInsGrid[r][0] = '\0';
        for (int i = 0; i < 13; i++) prevInsLine[i][0] = '\0';
        insCacheInit = true;
        compassBaseDrawn = false;   // screen was cleared; must redraw compass
circle
        compassNeedleDrawn = false;
        insScreenInit = true;
    }
    char grid[MAP_H][MAP_W + 1];
    for (int r = 0; r < MAP_H; r++) {
        for (int c = 0; c < MAP_W; c++) grid[r][c] = ' ';
        grid[r][MAP_W] = '\0';
    }
    float ox = startSet ? startX : 0.0f;
    float oy = startSet ? startY : 0.0f;
    float centerX = displayX_m;
```

```
    float centerY = displayY_m;
    float mPerCellX = MAP_SPAN_M / (MAP_W - 1);
    float mPerCellY = MAP_SPAN_M / (MAP_H - 1);
    int oCol = min(MAP_W - 1, (MAP_W / 2) + 2), oRow = MAP_H / 2;
    float crx = displayX_m - ox, cry = displayY_m - oy;
    int xCol = oCol;
    int xRow = min(MAP_H - 1, (MAP_H / 2) + 1);
    bool movedFromStart = fabsf(crx) > 0.35f || fabsf(cry) > 0.35f;
    if (startSet) {
        for (int i = 0; i < trackCount; i++) {
            int idx = (trackHead - trackCount + i + TRACK_MAX) % TRACK_MAX;
            float rx = track[idx].x - centerX, ry = track[idx].y - centerY;
            int col = oCol + (int)roundf(rx / mPerCellX);
            int row = oRow - (int)roundf(ry / mPerCellY);
            if (row >= 0 && row < MAP_H && col >= 0 && col < MAP_W)
grid[row][col] = '.';
        }
        int startCol = oCol + (int)roundf((startX - centerX) / mPerCellX);
        int startRow = oRow - (int)roundf((startY - centerY) / mPerCellY);
        if (startRow >= 0 && startRow < MAP_H && startCol >= 0 && startCol <
MAP_W) {
            grid[startRow][startCol] = 'O';
        }
        if (movedFromStart && xRow >= 0 && xRow < MAP_H && xCol >= 0 &&
xCol < MAP_W) grid[xRow][xCol] = 'X';
        for (int i = 0; i < victimCount; i++) {
            float rx = victims[i].x - centerX, ry = victims[i].y - centerY;
            int col = oCol + (int)roundf(rx / mPerCellX);
            int row = oRow - (int)roundf(ry / mPerCellY);
            if (row >= 0 && row < MAP_H && col >= 0 && col < MAP_W)
grid[row][col] = 'V';
        }
    }
    lcdBeginAccess();
    tft.setTextSize(1);
    tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
    for (int r = 0; r < MAP_H; r++) {
        if (!insCacheInit || strcmp(grid[r], prevInsGrid[r]) != 0) {
            tft.setCursor(2, 24 + r * 8);
            tft.print(grid[r]);
            strcpy(prevInsGrid[r], grid[r]);
        }
    }
    static int prevCompassHeading = -999;
    static int prevCompassTipX = 0;
    static int prevCompassTipY = 0;
```

```
    const int compassCx = 292;
    const int compassCy = 46;
    const int compassR = 14;
    int compassHeadingInt = compassValid ? (int)roundf(compassHeadingDeg *
10.0f) : -1;
    if (!compassBaseDrawn) {
        tft.fillRect(compassCx - 18, compassCy - 18, 36, 36, ILI9341_BLACK);
        tft.drawCircle(compassCx, compassCy, compassR, ILI9341_DARKGREY);
        tft.setTextColor(TILE_ORANGE, ILI9341_BLACK);
        tft.setCursor(compassCx - 3, compassCy - 24);
        tft.print("N");
        tft.setCursor(compassCx - 3, compassCy + 16);
        tft.print("S");
        tft.setCursor(compassCx - 21, compassCy - 3);
        tft.print("W");
        tft.setCursor(compassCx + 16, compassCy - 3);
        tft.print("E");
        compassBaseDrawn = true;
    }
    if (!insCacheInit || prevCompassHeading != compassHeadingInt) {
        if (compassNeedleDrawn) {
            tft.drawLine(compassCx, compassCy, prevCompassTipX,
prevCompassTipY, ILI9341_BLACK);
            tft.fillCircle(compassCx, compassCy, 2, ILI9341_BLACK);
        }
        if (compassValid) {
            float ang = compassHeadingDeg * DEG_TO_RAD;
            int tipX = compassCx + (int)roundf(sinf(ang) * (compassR - 2));
            int tipY = compassCy - (int)roundf(cosf(ang) * (compassR - 2));
            tft.drawLine(compassCx, compassCy, tipX, tipY, ILI9341_CYAN);
            tft.fillCircle(compassCx, compassCy, 2, ILI9341_WHITE);
            prevCompassTipX = tipX;
            prevCompassTipY = tipY;
            compassNeedleDrawn = true;
        } else {
            tft.fillCircle(compassCx, compassCy, 2, ILI9341_DARKGREY);
            compassNeedleDrawn = false;
        }
        prevCompassHeading = compassHeadingInt;
    }
    float rollNow = rollDegNowAbs;
    float pitchNow = pitchDegNowAbs;
    char lines[13][80];
    for (int i = 0; i < 13; i++) snprintf(lines[i], sizeof(lines[i]), " ");
    snprintf(lines[0], sizeof(lines[0]), "%s R:%5.1f P:%5.1f Y:%5.1f ",
gpsValidNow ? "GPS:FIX" : "GPS:DR", rollNow, pitchNow, yawDegDisp);
```

```
    snprintf(lines[1], sizeof(lines[1]), "EAST:%6.2f NORTH:%6.2f", crx, cry);
    float altitudeDisplayM = (isfinite(altitudeM) && isfinite(altitudeZeroM)) ?
(altitudeM - altitudeZeroM) : altitudeM;
    if (isfinite(altitudeDisplayM) && isfinite(altitudeZeroM)) snprintf(lines[2],
sizeof(lines[2]), "Alt:%7.2f m B:%7.2f m ", altitudeDisplayM, altitudeZeroM);
    else if (isfinite(altitudeDisplayM)) snprintf(lines[2], sizeof(lines[2]), "Alt:%7.2f m
B: --- ", altitudeDisplayM);
    else snprintf(lines[2], sizeof(lines[2]), "Alt: --- B: --- ");
    if (isfinite(insAmbientDisplayC) && isfinite(pressureHpa)) snprintf(lines[3],
sizeof(lines[3]), "Amb:%5.1fF P:%7.1fhPa ", cToF(insAmbientDisplayC),
pressureHpa);
    else snprintf(lines[3], sizeof(lines[3]), "Amb: --- P: --- ");
    float uwbRawM;
    uint32_t uwbRawMs;
    bool uwbRawValid;
    float uwbLastGoodSnap;
    noInterrupts();
    uwbRawM = uwbLatestDistanceM;
    uwbRawMs = uwbLatestUpdateMs;
    uwbRawValid = uwbLatestValid;
    uwbLastGoodSnap = uwbLastGoodDistanceM;
    interrupts();
    bool uwbHasReading = uwbRawValid && isfinite(uwbRawM) && ((millis() -
uwbRawMs) < UWB_DISPLAY_HOLD_MS);
    // Fall back to last-ever-good distance when UWB is off or hold has expired.
    float uwbDisplayM = uwbHasReading ? uwbRawM : uwbLastGoodSnap;
    bool uwbHasAny = isfinite(uwbDisplayM);
    char uwbShort[12];
    if (uwbHasAny) snprintf(uwbShort, sizeof(uwbShort), "%5.2f", uwbDisplayM);
    else snprintf(uwbShort, sizeof(uwbShort), "---");
    char distRow[32];
    if (uwbStartPending) {
        snprintf(distRow, sizeof(distRow), "FD:---m U:---m");
    } else if (isfinite(fusedDistanceM) && uwbHasAny) {
        snprintf(distRow, sizeof(distRow), "FD:%0.2fm U:%0.2fm", fusedDistanceM,
uwbDisplayM);
    } else if (isfinite(fusedDistanceM)) {
        snprintf(distRow, sizeof(distRow), "FD:%0.2fm U:---m", fusedDistanceM);
    } else if (uwbHasAny) {
        snprintf(distRow, sizeof(distRow), "FD:---m U:%0.2fm", uwbDisplayM);
    } else {
        snprintf(distRow, sizeof(distRow), "FD:---m U:---m");
    }
    snprintf(lines[4], sizeof(lines[4]), "%-28s", distRow);
    if (gpsValidNow) {
        snprintf(lines[5], sizeof(lines[5]), "LAT:%9.5f LON:%9.5f", lastLat, lastLon);
```

```
    } else {
        snprintf(lines[5], sizeof(lines[5]), "LAT: ---   LON: ---");
    }
    snprintf(lines[6], sizeof(lines[6]), "EAST V:%5.2f NORTH V:%5.2f", vE, vN);
    if (!gpsValidNow && victimLogCount == 0) {
        snprintf(lines[7], sizeof(lines[7]), "INDOOR UWB:%s E:%5.1f N:%5.1f",
                 uwbShort,
                 crx,
                 cry);
    }
    if (victimLogCount > 0) {
        int idx = (victimLogHead - 1 + VICTIM_LOG_MAX) % VICTIM_LOG_MAX;
        VictimSnapshot &s = victimLog[idx];
        if (s.gpsFix) {
            snprintf(lines[8], sizeof(lines[8]), "VICTIM:%d LAT:%0.5f LON:%0.5f",
victimCount, s.lat, s.lon);
        } else {
            snprintf(lines[8], sizeof(lines[8]), "VICTIM:%d UWB:%s E:%5.1f N:%5.1f",
                     victimCount,
                     uwbShort,
                     s.e_m,
                     s.n_m);
        }
    }
    tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
    for (int i = 0; i < 13; i++) {
        if (!insCacheInit || strcmp(lines[i], prevInsLine[i]) != 0) {
            tft.setCursor(2, INS_INFO_Y + i * INS_INFO_DY);
            tft.print(lines[i]);
            strcpy(prevInsLine[i], lines[i]);
        }
    }
    lcdEndAccess();
  insCacheInit = true;
}

static constexpr int SCAN_CARD_Y = 52;
static constexpr int SCAN_CARD_W = 96;
static constexpr int SCAN_CARD_H = 72;
static constexpr int SCAN_STATUS_Y = 136;
static constexpr int SCAN_STATUS_H = 96;
static constexpr int SCAN_STATUS_LINE_DY = 12;

static void drawScanBase() {

    lcdBeginAccess();
```

```
    tft.fillScreen(ILI9341_BLACK);
    tft.drawRect(0, 28, 320, 12, ILI9341_DARKGREY);
    tft.drawRoundRect(8,   SCAN_CARD_Y, SCAN_CARD_W, SCAN_CARD_H,
8, TILE_ORANGE);
    tft.drawRoundRect(112, SCAN_CARD_Y, SCAN_CARD_W, SCAN_CARD_H,
8, ILI9341_CYAN);
    tft.drawRoundRect(216, SCAN_CARD_Y, SCAN_CARD_W, SCAN_CARD_H,
8, ILI9341_GREEN);
    tft.drawRoundRect(8, SCAN_STATUS_Y, 304, SCAN_STATUS_H, 8,
ILI9341_DARKGREY);
    tft.setTextSize(1);
    tft.setTextColor(TILE_ORANGE,    ILI9341_BLACK); tft.setCursor(34,
SCAN_CARD_Y + 10); tft.print("HR AVG");
    tft.setTextColor(ILI9341_CYAN,   ILI9341_BLACK); tft.setCursor(138,
SCAN_CARD_Y + 10); tft.print("BR AVG");
    tft.setTextColor(ILI9341_GREEN,  ILI9341_BLACK); tft.setCursor(238,
SCAN_CARD_Y + 10); tft.print("DIST");
    tft.setTextColor(ILI9341_DARKGREY, ILI9341_BLACK); tft.setCursor(122,
SCAN_STATUS_Y - 10); tft.print("SCAN STATUS");
    lcdEndAccess();
    topBarDirty = true;
    drawTopBar("SCAN", true, scanModeShowsMarkButton());
    for (int i = 0; i < 10; i++) prevScanLine[i][0] = '\0';
    prevScanPhaseLabel[0] = '\0';
    prevScanPhaseTimer[0] = '\0';
    prevScanProgressFill = 0;
    scanScreenInit = true;
}

static void renderScanThermalReview() {
    if (!scanThermalReviewInit) {
        // Don't nuke the human-detected alert clip here. If CLIP_HUMAN is
        // already playing (SCAN_IR fired the alert on first detection) we want
        // it to finish so the scan thermal screen behaves exactly like the
        // stand-alone THERMAL screen. Only kill non-alert audio such as the
        // loading clip that may still be tailing off from SCAN_LOADING.
        bool alertPlayingNow =
            alertTriggered &&
            currentClip == CLIP_HUMAN &&
            (audioState == AUDIO_STARTING ||
             audioState == AUDIO_PLAYING  ||
             audioState == AUDIO_FINISHING);
        if (!alertPlayingNow) {
            stopAudioNow();
            resetAudioState();
        }
```

```
        lcdBeginAccess();
        tft.fillScreen(ILI9341_BLACK);
        lcdEndAccess();
        prevThermalValid = false;
        thermalHudInit = false;
        topBarDirty = true;
        scanThermalReviewInit = true;
    }
    runThermal();
    drawTopBar("SCAN", true, scanModeShowsMarkButton());
}

static void renderScanNavReview() {
    if (!scanNavReviewInit) {
        stopAudioNow();
        resetAudioState();
        lcdBeginAccess();
        tft.fillScreen(ILI9341_BLACK);
        lcdEndAccess();
        insScreenInit = false;
        insCacheInit = false;
        topBarDirty = true;
        scanNavReviewInit = true;
    }
    runINS();
    drawTopBar("SCAN", true, scanModeShowsMarkButton());
}

void runScan() {
    // Copied verbatim from runThermal()'s alert guard (see ~line 3838).
    // While the HUMAN clip is playing we MUST keep loop() tight so
    // serviceAudio() is called often enough to keep the I2S DMA buffer fed;
    // one full scan tick (MLX capture ~100 ms + radar UART + UWB + fusion +
    // LCD render) between serviceAudio() calls is more than the decoder's
    // buffer depth and is what was making "Human detected" stutter.
    //
    // Draw the HUMAN DETECTED banner once and return — identical
behaviour to
    // the stand-alone THERMAL screen so the clip plays cleanly.
    static bool scanHumanOverlayDrawn = false;
    if (alertTriggered &&
        currentClip == CLIP_HUMAN &&
        (audioState == AUDIO_STARTING ||
         audioState == AUDIO_PLAYING  ||
         audioState == AUDIO_FINISHING)) {
        if (!scanHumanOverlayDrawn) {
```

```
        lcdBeginAccess();
        tft.fillRoundRect(40, 100, 240, 40, 8, ILI9341_GREEN);
        tft.setTextSize(2);
        tft.setTextColor(ILI9341_BLACK, ILI9341_GREEN);
        tft.setCursor(52, 114);
        tft.print("HUMAN DETECTED");
        lcdEndAccess();
        scanHumanOverlayDrawn = true;
    }
    return;
}
// Alert finished — mirror runThermal()'s minimal cleanup: only invalidate
// the thermal image + HUD caches so they repaint over the green banner on
// the next frame. Do NOT touch scanThermalReviewInit / scanScreenInit /
// top-bar / scan-line caches here — the green banner only covers the
// thermal image area (y=100..140), so the cards, toolbar and top bar are
// still valid. Invalidating the review-init flag would cause
// renderScanThermalReview() to run its fillScreen(BLACK) init branch on
// the next frame, producing the visible flash we used to have.
if (scanHumanOverlayDrawn) {
    scanHumanOverlayDrawn = false;
    prevThermalValid = false;
    thermalHudInit = false;
    // Pin the render path to the thermal review screen as soon as the
    // HUMAN clip ends. The scan engine is mid-flight (still in SCAN_IR /
    // POSITION / FUSION or transitioning toward SCAN_DONE) and on the
    // very frame the banner disappears it's possible — depending on what
    // the sensors looked like while the state machine was frozen — for
    // updateScanMode() to advance into a state where both the
    // "scanReviewSequenceActive" branch and the "state is IR/POS/FUSION"
    // branch miss for a single frame, and we fall through to the scan
    // summary/cards render at the bottom of runScan(). That single frame
    // is what was flashing "summary" right as the banner cleared.
    // Latching the review now forces renderScanThermalReview() to own
    // the screen from the instant the clip finishes until the operator
    // presses NEXT, exactly mirroring the stand-alone THERMAL screen
    // (which has nothing to fall through to). The SCAN_DONE block below
    // is idempotent — when the engine eventually reaches DONE it sees
    // these already set and the latch flag trips normally.
    if (currentMode == SCAN_MODE && scanRt.active) {
        scanReviewSequenceActive = true;
        scanReviewScreen = SCAN_REVIEW_THERMAL;
    }
}
```

```
    // NOTE: we intentionally do NOT short-circuit runScan() on
CLIP_TOOCLOSE
    // here. serviceTooClose() runs AFTER loop()'s tight-pump (see comment
    // there) which means on the iteration that fires the clip, runScan() MUST
    // still run so updateProximityReading() keeps proxTooCloseLatched fresh.
    // Without that refresh, the pump would exit on AUDIO_DONE with the flag
    // still latched and immediately restart the clip forever. This mirrors the
    // PROXIMITY screen's original working behaviour.

    if (!scanScreenInit) {
        drawScanBase();
    }
    updateProximityReading();
    static ScanMode::ScanState prevScanState = ScanMode::SCAN_IDLE;
    if (scanRt.state == ScanMode::SCAN_IR && prevScanState !=
ScanMode::SCAN_IR) {
        thermalHumanDetected = false;
        thermalFrameValid = false;
        prevThermalValid = false;
        ThermalHuman::initState(thermalDetectorState);
        memset(&thermalDetectorFeatures, 0, sizeof(thermalDetectorFeatures));
        memset(&thermalDetectorScores, 0, sizeof(thermalDetectorScores));
    }
    prevScanState = scanRt.state;

    // Keep the thermal detector running across every phase where the scan
    // thermal screen is on-screen. Previously this was gated to SCAN_IR only,
    // but SCAN_IR exits on the first positive frame — so the 2-frame alert
    // debounce in updateThermalDetection() never reached frame 2 and the
    // HUMAN clip never played on the scan thermal screen. Running it through
    // POSITION / FUSION / DONE (and the post-scan thermal review) lets the
    // detector confirm and fire triggerAlert() exactly like the stand-alone
    // THERMAL screen.
    // Same gate as alertAllowedForCurrentScreen(): thermal detection only
    // runs while the thermal map is actually on-screen. Dropping the bare
    // SCAN_DONE clause stops the detector from confirming + firing the
    // HUMAN clip after the operator has pressed NEXT past the thermal
    // review onto NAV or the final SUMMARY screen, which is what was
    // causing the green banner to pop up over the summary and also pinning
    // scanReviewScreen back to THERMAL (the "thermal reopens after
    // completion" symptom).
    bool thermalScreenVisible =
        scanRt.state == ScanMode::SCAN_IR       ||
        scanRt.state == ScanMode::SCAN_POSITION ||
        scanRt.state == ScanMode::SCAN_FUSION   ||
```

```
        (scanReviewSequenceActive && scanReviewScreen ==
SCAN_REVIEW_THERMAL);
    if (thermalScreenVisible) {
        updateThermalDetection();
        // Keep the IR confidence in the scan runtime synced with the live
        // thermal detector once we're past SCAN_IR. latchIrResult() froze
        // rt.ir.confidenceScore the instant SCAN_IR exited (first confirmed
        // detection), and fuseResults() wouldn't run again after the fusion
        // hold — so the "Final %" and Decision shown on the scan thermal
        // review screen used to stay stuck at that snapshot value even
        // though the live thermal detector kept climbing while the subject
        // stood in the MLX frame. Pushing the live value back into the
        // runtime and re-running fuseResults() here makes the Final % and
        // Decision text track the same C: value the operator sees in the
        // HUD, so the longer the person stays in frame the higher it goes.
        if (scanRt.active &&
            scanRt.state != ScanMode::SCAN_IDLE &&
            scanRt.state != ScanMode::SCAN_INIT &&
            scanRt.state != ScanMode::SCAN_RADAR &&
            scanRt.state != ScanMode::SCAN_IR) {
            scanRt.ir.confidenceScore      = clampf((float)thermalConfidencePct /
100.0f, 0.0f, 1.0f);
            scanRt.ir.humanPresenceScore   =
clampf(thermalDetectorScores.human_presence_score, 0.0f, 1.0f);
            scanRt.ir.humanLike            = thermalHumanDetected;
            ScanMode::fuseResults(scanRt, scanCfg);
        }
    }
    radarPoll();
    if (uwbActive && uwbInitOk) {
        pollUWB();
        updateFusedDistance();
    }

    ScanMode::ScanSensorInputs in{};
    in.ussTooClose = proxTooCloseLatched;
    in.radarMotionDetected = radarHuman;
    in.radarPresenceDetected = radarHuman;
    in.radarBreathingValid = isfinite(radarBreathDisplayBpm);
    in.radarHeartValid = isfinite(radarHeartDisplayBpm);
    in.radarBreathingBpm = radarBreathDisplayBpm;
    in.radarHeartRateBpm = radarHeartDisplayBpm;
    in.radarMotionPresenceScore = radarHuman ? 0.80f : 0.0f;
    in.radarSignalQuality = clampf(radarPeakScore, 0.0f, 1.0f);
    in.radarMotionStability = radarHuman ? 0.70f : 0.0f;
    in.radarBreathingVariance = isfinite(radarBreathDisplayBpm) ? 0.20f : 1.0f;
```

```
    in.radarHeartVariance = isfinite(radarHeartDisplayBpm) ? 0.20f : 1.0f;
    in.radarBreathingConsistency = isfinite(radarBreathDisplayBpm) ? 0.75f : 0.0f;
    in.radarHeartConsistency = isfinite(radarHeartDisplayBpm) ? 0.70f : 0.0f;
    in.irHumanLike = thermalHumanDetected;
    in.irHumanPresenceScore = thermalDetectorScores.human_presence_score;
    in.irConfidenceScore = thermalDetectorScores.confidence;
    in.irCategoryScores = thermalDetectorScores.class_scores;
    in.irBBoxX = thermalDetectorFeatures.bbox_x;
    in.irBBoxY = thermalDetectorFeatures.bbox_y;
    in.irBBoxW = thermalDetectorFeatures.bbox_w;
    in.irBBoxH = thermalDetectorFeatures.bbox_h;
    in.irCentroidX = thermalDetectorFeatures.centroid_x;
    in.irCentroidY = thermalDetectorFeatures.centroid_y;
    in.irAspectRatio = thermalDetectorFeatures.aspect_ratio;
    in.irHotPixelCount = thermalDetectorFeatures.hot_pixel_count;
    in.irMeanTargetTemp = thermalDetectorFeatures.mean_target_temp;
    in.irMaxTargetTemp = thermalDetectorFeatures.max_target_temp;
    in.irContrastAboveBackground =
thermalDetectorFeatures.contrast_above_local_background;
    in.irTopCategory =
ThermalHuman::className(thermalDetectorScores.best_class);
    in.irDebug = thermalDetectorFeatures.debug;
    in.uwbValid = uwbLatestValid;
    in.uwbRangeM = uwbLatestDistanceM;
    in.gnssValid = gpsValidNow;
    in.lat = lastLat;
    in.lon = lastLon;
    in.insValid = imuReady;
    in.relX_m = startSet ? (displayX_m - startX) : displayX_m;
    in.relY_m = startSet ? (displayY_m - startY) : displayY_m;
    in.relZ_m = (isfinite(altitudeM) && isfinite(altitudeZeroM)) ? (altitudeM -
altitudeZeroM) : altitudeM;
    in.rollDeg = rollDegDisp;
    in.pitchDeg = pitchDegDisp;
    in.yawDeg = yawDegDisp;
    in.positionConfidence = (imuReady ? 0.45f : 0.0f) + (gpsValidNow ? 0.35f :
0.0f) + (uwbLatestValid ? 0.20f : 0.0f);

    if (scanRt.active) {
        ScanMode::ScanCallbacks cb{};
        ScanMode::updateScanMode(scanRt, scanCfg, in, cb, millis());
    }

    if (scanRt.state == ScanMode::SCAN_DONE
&& !scanReviewCompletionLatched) {
        // Let the human-detected clip keep playing across SCAN_DONE so the
```

```
        // scan thermal review screen feels identical to the THERMAL screen.
        bool alertPlayingNow =
            alertTriggered &&
            currentClip == CLIP_HUMAN &&
            (audioState == AUDIO_STARTING ||
             audioState == AUDIO_PLAYING  ||
             audioState == AUDIO_FINISHING);
        if (!alertPlayingNow) {
            stopAudioNow();
            resetAudioState();
        }
        scanReviewCompletionLatched = true;
        scanReviewSequenceActive = true;
        scanReviewScreen = SCAN_REVIEW_THERMAL;
        // Don't force a full redraw if we're already sitting on the thermal
        // review: scan mode now stays on this screen from SCAN_IR through
        // SCAN_DONE, so resetting the init flag would blank and redraw the
        // panel (and cause a visible flash) right as the scan completes.
        scanNavReviewInit = false;
        toolbarSelection = TOOLBAR_START;
        topBarDirty = true;
    }

    if (scanReviewSequenceActive) {
        if (scanReviewScreen == SCAN_REVIEW_THERMAL) {
            renderScanThermalReview();
            return;
        }
        if (scanReviewScreen == SCAN_REVIEW_NAV) {
            renderScanNavReview();
            return;
        }
    }

    // Once the radar phase finishes, keep the operator pinned to the thermal
    // review screen through IR, POSITION, FUSION and DONE. The downstream
phases
    // exit early (IR on detection, POSITION on first valid fix, FUSION after its
    // 1.2s hold) and used to flash nav -> summary -> thermal in rapid succession.
    // Staying on thermal lets the operator press NEXT to step through screens
    // manually instead of watching them auto-flip.
    if (scanRt.active &&
        (scanRt.state == ScanMode::SCAN_IR ||
         scanRt.state == ScanMode::SCAN_POSITION ||
         scanRt.state == ScanMode::SCAN_FUSION)) {
        renderScanThermalReview();
```

```
        return;
    }

    char lines[10][48];
    char hrBuf[16];
    char brBuf[16];
    char distBuf[16];
    char phaseLabelBuf[40];
    char phaseTimerBuf[12];
    float cardDistM = fusedDist(radarCurrentDistanceM());
    const int irConfidencePct = constrain((int)lroundf(100.0f *
scanDisplayConfidence()), 0, 100);

    if (isfinite(scanRt.radar.heartRateBpm)) snprintf(hrBuf, sizeof(hrBuf), "%.1f",
scanRt.radar.heartRateBpm);
    else strncpy(hrBuf, "--", sizeof(hrBuf) - 1);
    hrBuf[sizeof(hrBuf) - 1] = '\0';
    if (isfinite(scanRt.radar.breathingBpm)) snprintf(brBuf, sizeof(brBuf), "%.1f",
scanRt.radar.breathingBpm);
    else strncpy(brBuf, "--", sizeof(brBuf) - 1);
    brBuf[sizeof(brBuf) - 1] = '\0';
    if (isfinite(cardDistM)) snprintf(distBuf, sizeof(distBuf), "%.2f", cardDistM);
    else strncpy(distBuf, "--", sizeof(distBuf) - 1);
    distBuf[sizeof(distBuf) - 1] = '\0';

    float phaseRemainingS = 0.0f;
    uint32_t phaseBudgetMs = 0;
    switch (scanRt.state) {
        case ScanMode::SCAN_INIT: phaseBudgetMs = scanCfg.initTimeoutMs;
break;
        case ScanMode::SCAN_RADAR: phaseBudgetMs =
scanCfg.radarMaxDurationMs; break;
        case ScanMode::SCAN_IR: phaseBudgetMs = scanCfg.irMaxDurationMs;
break;
        case ScanMode::SCAN_POSITION: phaseBudgetMs =
scanCfg.positionMaxDurationMs; break;
        case ScanMode::SCAN_FUSION: phaseBudgetMs =
scanCfg.fusionHoldMs; break;
        default: phaseBudgetMs = 0; break;
    }
    if (phaseBudgetMs > 0 && scanRt.stateStartMs > 0) {
        uint32_t elapsedMs = millis() - scanRt.stateStartMs;
        phaseRemainingS = max(0.0f, (float)(phaseBudgetMs -
min<uint32_t>(phaseBudgetMs, elapsedMs)) / 1000.0f);
    }
```

```
    const char* phaseDetail = "-";
    switch (scanRt.state) {
        case ScanMode::SCAN_RADAR: phaseDetail = "Collecting radar vitals";
break;
        case ScanMode::SCAN_IR: phaseDetail = "Thermal human check"; break;
        case ScanMode::SCAN_POSITION: phaseDetail = "UWB/GNSS/INS
position"; break;
        case ScanMode::SCAN_FUSION: phaseDetail = "Fusing sensor verdict";
break;
        case ScanMode::SCAN_DONE: phaseDetail = "Scan complete"; break;
        case ScanMode::SCAN_ABORT: phaseDetail = "Scan aborted"; break;
        default: phaseDetail = "Preparing sensors"; break;
    }

    lines[0][0] = '\0';
    lines[1][0] = '\0';
    lines[2][0] = '\0';
    if (!scanRt.active) {
        snprintf(lines[3], sizeof(lines[3]), "State: %s",
ScanMode::stateName(scanRt.state));
        snprintf(phaseLabelBuf, sizeof(phaseLabelBuf), "Phase: Press START");
        phaseTimerBuf[0] = '\0';
        strncpy(lines[4], phaseLabelBuf, sizeof(lines[4]) - 1);
        lines[4][sizeof(lines[4]) - 1] = '\0';
        snprintf(lines[5], sizeof(lines[5]), "Radar ready  UWB:%s",
                uwbInitOk ? "Y" : (uwbStartPending ? "INIT" : "N"));
        snprintf(lines[6], sizeof(lines[6]), "Thermal ready  INS:%s", imuReady ? "Y" :
"N");
        snprintf(lines[7], sizeof(lines[7]), "Scan runs radar -> thermal -> pos");
        if (gpsValidNow) {
            snprintf(lines[8], sizeof(lines[8]), "Pos GPS:Y UWB:%s INS:%s",
                    uwbInitOk ? "Y" : "N",
                    imuReady ? "Y" : "N");
            snprintf(lines[9], sizeof(lines[9]), "LAT:%.5f LON:%.5f", lastLat, lastLon);
        } else {
            snprintf(lines[8], sizeof(lines[8]), "Pos INDOOR UWB:%s INS:%s",
                    uwbInitOk ? "Y" : "N",
                    imuReady ? "Y" : "N");
            snprintf(lines[9], sizeof(lines[9]), "E:%.1f N:%.1f Y:%.0f", displayX_m,
displayY_m, yawDegDisp);
        }
    } else {
        snprintf(lines[3], sizeof(lines[3]), "State: %s",
ScanMode::stateName(scanRt.state));
        snprintf(phaseLabelBuf, sizeof(phaseLabelBuf), "Phase: %s", phaseDetail);
```

```
        if (phaseBudgetMs > 0) snprintf(phaseTimerBuf, sizeof(phaseTimerBuf),
"%.1fs", phaseRemainingS);
        else phaseTimerBuf[0] = '\0';
        strncpy(lines[4], phaseLabelBuf, sizeof(lines[4]) - 1);
        lines[4][sizeof(lines[4]) - 1] = '\0';
        // Show every sensor metric in the same 0-100% form so they are easy to
        // compare at a glance. The Final column maps 1:1 onto the Human /
        // Possible / No-human decision (>=70 / >=40 / <40) from fuseResults().
        const int radarPct = constrain((int)lroundf(100.0f *
scanRt.radar.radarConfidence), 0, 100);
        const int finalPct = constrain((int)lroundf(100.0f * scanDisplayConfidence()),
0, 100);
        snprintf(lines[5], sizeof(lines[5]), "Radar %d%%  Therm %d%%
Final %d%%",
                 radarPct, irConfidencePct, finalPct);
        snprintf(lines[6], sizeof(lines[6]), "Decision: %s",
scanRt.fusion.decisionText);
        snprintf(lines[7], sizeof(lines[7]), "IR:%s hot:%d max:%.1fF",
                 scanRt.ir.topCategory[0] ? scanRt.ir.topCategory : "-",
                 scanRt.ir.hotPixelCount, cToF(scanRt.ir.maxTargetTemp));
        if (scanRt.position.gnssValid) {
          snprintf(lines[8], sizeof(lines[8]), "Pos GPS:%s UWB:%s INS:%s",
                   scanRt.position.gnssValid ? "Y" : "N",
                   scanRt.position.uwbValid ? "Y" : "N",
                   scanRt.position.insValid ? "Y" : "N");
          snprintf(lines[9], sizeof(lines[9]), "LAT:%.5f LON:%.5f",
                   scanRt.position.lat, scanRt.position.lon);
        } else if (scanRt.position.uwbValid || scanRt.position.insValid) {
          snprintf(lines[8], sizeof(lines[8]), "Pos INDOOR UWB:%s INS:%s",
                   scanRt.position.uwbValid ? "Y" : "N",
                   scanRt.position.insValid ? "Y" : "N");
          snprintf(lines[9], sizeof(lines[9]), "E:%.1f N:%.1f Y:%.0f",
                   scanRt.position.relX_m, scanRt.position.relY_m,
scanRt.position.yawDeg);
        } else {
          snprintf(lines[8], sizeof(lines[8]), "Pos UWB:%s XY:%.1f %.1f Y:%.0f",
                   scanRt.position.uwbValid ? "Y" : "N",
                   scanRt.position.relX_m, scanRt.position.relY_m,
scanRt.position.yawDeg);
          snprintf(lines[9], sizeof(lines[9]), "%s",
                   scanRt.fusion.warning[0] ? scanRt.fusion.warning :
                   (scanRt.radar.warning[0] ? scanRt.radar.warning :
scanRt.position.debug));
        }
    }
```

```
    // Pause heavy LCD render while audio is active (mirrors runThermal() guard).
    static bool scanAlertOverlayDrawn = false;
    if (alertTriggered &&
        !scanReviewSequenceActive &&
        (scanRt.state == ScanMode::SCAN_IR ||
        (scanReviewSequenceActive && scanReviewScreen ==
SCAN_REVIEW_THERMAL)) &&
       (audioState == AUDIO_STARTING ||
        audioState == AUDIO_PLAYING  ||
        audioState == AUDIO_FINISHING)) {

       if (!scanAlertOverlayDrawn) {
          lcdBeginAccess();
          tft.fillRoundRect(40, 100, 240, 40, 8, ILI9341_GREEN);
          tft.setTextSize(2);
          tft.setTextColor(ILI9341_BLACK, ILI9341_GREEN);
          tft.setCursor(52, 114);
          tft.print("HUMAN DETECTED");
          lcdEndAccess();
          scanAlertOverlayDrawn = true;
       }
       return;
    }
    if (scanAlertOverlayDrawn) {
       scanAlertOverlayDrawn = false;
       for (int i = 0; i < 10; i++) prevScanLine[i][0] = '\0';
       prevScanPhaseLabel[0] = '\0';
       prevScanPhaseTimer[0] = '\0';
       prevScanProgressFill = 0;
    }
    // Pause heavy LCD render while non-alert audio is active (e.g. loading clip).
static bool scanAudioPaused = false;

bool blockingAudio =
    currentClip == CLIP_TOOCLOSE ||
    currentClip == CLIP_HUMAN ||
    currentClip == CLIP_SCAN;

if (blockingAudio &&
    (audioState == AUDIO_STARTING ||
     audioState == AUDIO_PLAYING ||
     audioState == AUDIO_FINISHING) &&
    !scanReviewSequenceActive &&
    scanRt.state != ScanMode::SCAN_IR &&
    scanRt.state != ScanMode::SCAN_POSITION &&
    scanRt.state != ScanMode::SCAN_DONE) {
```

```
    scanAudioPaused = true;
    return;
}
    if (scanAudioPaused) {
        for (int i = 0; i < 10; i++) prevScanLine[i][0] = '\0';
        prevScanPhaseLabel[0] = '\0';
        prevScanPhaseTimer[0] = '\0';
        prevScanProgressFill = 0;
        scanAudioPaused = false;
    }

    lcdBeginAccess();
    uint16_t fill = (uint16_t)((scanRt.progressPermille * 316UL) / 1000UL);
    if (fill != prevScanProgressFill) {
        tft.fillRect(2, 30, 316, 8, ILI9341_NAVY);
        if (fill > 0) tft.fillRect(2, 30, fill, 8, TILE_ORANGE);
        prevScanProgressFill = fill;
    }

    const int valueY = SCAN_CARD_Y + 32;
    tft.setTextSize(2);
    if (strcmp(hrBuf, prevScanLine[0]) != 0) {
        tft.fillRect(16, valueY, SCAN_CARD_W - 16, 24, ILI9341_BLACK);
        tft.setCursor(22, valueY);
        tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
        tft.print(hrBuf);
        strncpy(prevScanLine[0], hrBuf, sizeof(prevScanLine[0]) - 1);
        prevScanLine[0][sizeof(prevScanLine[0]) - 1] = '\0';
    }
    if (strcmp(brBuf, prevScanLine[1]) != 0) {
        tft.fillRect(120, valueY, SCAN_CARD_W - 16, 24, ILI9341_BLACK);
        tft.setCursor(126, valueY);
        tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
        tft.print(brBuf);
        strncpy(prevScanLine[1], brBuf, sizeof(prevScanLine[1]) - 1);
        prevScanLine[1][sizeof(prevScanLine[1]) - 1] = '\0';
    }
    if (strcmp(distBuf, prevScanLine[2]) != 0) {
        tft.fillRect(224, valueY, SCAN_CARD_W - 16, 24, ILI9341_BLACK);
        tft.setCursor(224, valueY);
        tft.setTextColor(ILI9341_WHITE, ILI9341_BLACK);
        tft.print(distBuf);
        strncpy(prevScanLine[2], distBuf, sizeof(prevScanLine[2]) - 1);
        prevScanLine[2][sizeof(prevScanLine[2]) - 1] = '\0';
    }
```

```
    tft.setTextSize(1);
    tft.setTextColor(ILI9341_DARKGREY, ILI9341_BLACK);
    tft.setCursor(38,  SCAN_CARD_Y + 56); tft.print("bpm");
    tft.setCursor(142, SCAN_CARD_Y + 56); tft.print("bpm");
    tft.setCursor(252, SCAN_CARD_Y + 56); tft.print("m");

    for (int i = 3; i < 10; i++) {
        if (i == 4) continue;
        if (strcmp(lines[i], prevScanLine[i]) != 0) {
            tft.fillRect(16, SCAN_STATUS_Y + 4 + (i - 3) *
SCAN_STATUS_LINE_DY, 288, 10, ILI9341_BLACK);
            tft.setCursor(16, SCAN_STATUS_Y + 4 + (i - 3) *
SCAN_STATUS_LINE_DY);
            tft.setTextColor((i == 4 || i == 6) ? ILI9341_CYAN : ILI9341_WHITE,
ILI9341_BLACK);
            tft.print(lines[i]);
            strncpy(prevScanLine[i], lines[i], sizeof(prevScanLine[i]) - 1);
            prevScanLine[i][sizeof(prevScanLine[i]) - 1] = '\0';
        }
    }
    const int phaseY = SCAN_STATUS_Y + 4 + (4 - 3) *
SCAN_STATUS_LINE_DY;
    if (strcmp(phaseLabelBuf, prevScanPhaseLabel) != 0) {
        tft.fillRect(16, phaseY, 220, 10, ILI9341_BLACK);
        tft.setCursor(16, phaseY);
        tft.setTextColor(ILI9341_CYAN, ILI9341_BLACK);
        tft.print(phaseLabelBuf);
        strncpy(prevScanPhaseLabel, phaseLabelBuf, sizeof(prevScanPhaseLabel)
- 1);
        prevScanPhaseLabel[sizeof(prevScanPhaseLabel) - 1] = '\0';
    }
    if (strcmp(phaseTimerBuf, prevScanPhaseTimer) != 0) {
        tft.fillRect(244, phaseY, 60, 10, ILI9341_BLACK);
        if (phaseTimerBuf[0]) {
            int timerX = 304 - ((int)strlen(phaseTimerBuf) * 6);
            if (timerX < 244) timerX = 244;
            tft.setCursor(timerX, phaseY);
            tft.setTextColor(ILI9341_CYAN, ILI9341_BLACK);
            tft.print(phaseTimerBuf);
        }
        strncpy(prevScanPhaseTimer, phaseTimerBuf,
sizeof(prevScanPhaseTimer) - 1);
        prevScanPhaseTimer[sizeof(prevScanPhaseTimer) - 1] = '\0';
    }
    lcdEndAccess();
}
```

```
// ================== BLE ==================
// Persistent BLE handler. Maintains the connection regardless of which
// operating mode is active; future work will route per-mode data through here.

class BleServerCallbacks : public NimBLEServerCallbacks {
    // NimBLE-Arduino v2 signatures — connInfo carries the connection handle.
    void onConnect(NimBLEServer* pServer, NimBLEConnInfo& connInfo)
override {
    bleConnHandles.push_back(connInfo.getConnHandle());
    bleState           = BLE_CONNECTED;
    bleNeedsRedraw     = true;
    bleConnectedMs     = millis();
    bleSendConnectedLog = true;
    NimBLEDevice::getAdvertising()->start();
}
    void onDisconnect(NimBLEServer* pServer, NimBLEConnInfo& connInfo, int
reason) override {
    uint16_t handle = connInfo.getConnHandle();
    bleConnHandles.erase(
        std::remove(bleConnHandles.begin(), bleConnHandles.end(), handle),
        bleConnHandles.end()
    );
    bleNeedsRedraw = true;
    if (bleConnHandles.empty()) {
        bleState = BLE_DISCONNECTED;
    }
    NimBLEDevice::getAdvertising()->start();
}
};

void setupBLE() {
    // MTU is NOT set here intentionally. Android 5.0+ automatically requests
    // MTU 512 during connection, and NimBLE accepts it — no explicit request
    // needed. Calling setMTU() before init() corrupts NimBLE heap state.
    NimBLEDevice::init("DWM3000-SAR");
    NimBLEDevice::setMTU(512);
    bleServer = NimBLEDevice::createServer();
    bleServer->setCallbacks(new BleServerCallbacks());

    NimBLEService* svc = bleServer->createService("680c21d9-c946-4c1f-9c11-
baa1c21329e7");

    // Log / command channel — app writes commands, ESP32 sends UTF-8
status strings
    bleCharacteristic = svc->createCharacteristic(
```

```
        "003bbdf2-c634-4b3d-ab56-7ec889b89a37",
        NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY |
NIMBLE_PROPERTY::WRITE
    );

    // Thermal frame channel — ESP32 sends chunked binary thermal frames
    bleCharThermal = svc->createCharacteristic(
        BLE_THERMAL_CHAR_UUID,
        NIMBLE_PROPERTY::NOTIFY
    );

    svc->start();

    // Advertise the Service UUID so Android can find the device by UUID scan
filter
    NimBLEDevice::getAdvertising()->addServiceUUID(svc->getUUID());
    // Advertising is started on demand when the user presses the BLE tile.
}

// Send a UTF-8 string to the connected Android app via BLE notification.
void bleSendLog(const char* msg) {
    if (bleState != BLE_CONNECTED || !bleCharacteristic) return;
    bleCharacteristic->setValue((uint8_t*)msg, strlen(msg));
    for (uint16_t handle : bleConnHandles) {
        bleCharacteristic->notify(handle);
    }
}

// Build and send one 776-byte thermal frame over BLE in chunked notifications.
//
// Frame layout (776 bytes):
//   [0..3]   float  minTempC  (little-endian)
//   [4..7]   float  maxTempC  (little-endian)
//   [8..775] uint8  pixels[768] — each mapped 0-255 across [minTempC,
maxTempC]
//
// Each BLE packet: [chunk_idx uint8][total_chunks uint8][data bytes...]
// The Android app reassembles chunks by index, then parses the frame.
void bleSendThermalFrame() {
    if (bleState != BLE_CONNECTED || !bleCharThermal || !thermalFrameValid)
return;

    float minC = mlxFilt[0], maxC = mlxFilt[0];
    for (int i = 1; i < 768; i++) {
        if (mlxFilt[i] < minC) minC = mlxFilt[i];
        if (mlxFilt[i] > maxC) maxC = mlxFilt[i];
```

```
    }
    float range = maxC - minC;
    if (range < 0.1f) range = 0.1f;

    // Build quantized frame
    static uint8_t prevFrame[768] = {0};
    static bool prevFrameValid = false;
    static uint8_t frame[776];
    static uint8_t packet[2 + BLE_THERMAL_CHUNK_DATA];

    memcpy(frame + 0, &minC, 4);
    memcpy(frame + 4, &maxC, 4);
    for (int i = 0; i < 768; i++) {
        float norm = (mlxFilt[i] - minC) / range;
        if (norm < 0.0f) norm = 0.0f;
        if (norm > 1.0f) norm = 1.0f;
        frame[8 + i] = (uint8_t)(norm * 255.0f);
    }

    // Skip send if frame hasn't changed meaningfully
    if (prevFrameValid) {
        int diffCount = 0;
        for (int i = 0; i < 768; i++) {
            if (abs((int)frame[8+i] - (int)prevFrame[i]) > 3) diffCount++;
        }
        if (diffCount < 5) return;  // skip nearly identical frames
    }
    memcpy(prevFrame, frame + 8, 768);
    prevFrameValid = true;

    const int FRAME_LEN    = 776;
    const int total_chunks = (FRAME_LEN + BLE_THERMAL_CHUNK_DATA - 1)
/ BLE_THERMAL_CHUNK_DATA;

    for (int idx = 0; idx < total_chunks; idx++) {
        int offset   = idx * BLE_THERMAL_CHUNK_DATA;
        int data_len = FRAME_LEN - offset;
        if (data_len > BLE_THERMAL_CHUNK_DATA) data_len =
BLE_THERMAL_CHUNK_DATA;
        packet[0] = (uint8_t)idx;
        packet[1] = (uint8_t)total_chunks;
        memcpy(packet + 2, frame + offset, data_len);
        bleCharThermal->setValue(packet, 2 + data_len);
        for (uint16_t handle : bleConnHandles) {
            bleCharThermal->notify(handle);
        }
```

```
  }
}

// ── RADAR mode BLE packet ──────────────────────────────────────────────
//
// Sent on bleCharacteristic (UTF-8 string notify) at ~5 Hz while in PROXIMITY
// mode with BLE connected.
//
// Packet format  (pipe-delimited):
//
RADAR|<yaw_deg>|<rdist_m>|<human>|<br_bpm>|<hr_bpm>|<uss_cm>|<toocl
ose>
//
//   yaw_deg  : float 1dp — IMU heading (0–360°); sweep-line angle for polar map
//   rdist_m  : float 2dp — fused radar target distance (m); "nan" if none
//   human    : 0 or 1   — radarHuman detection flag; draw dot when 1
//   br_bpm   : float 1dp — fused breathing rate (BPM); "nan" if unavailable
//   hr_bpm   : float 1dp — fused heart rate (BPM); "nan" if unavailable
//   uss_cm   : float 1dp — ultrasonic proximity (cm); "nan" if no reading
//   tooclose : 0 or 1   — proxTooCloseLatched safety flag
//
// Example: "RADAR|45.2|1.23|1|15.2|72.1|45.3|0"
void bleSendRadarData() {
  if (bleState != BLE_CONNECTED || !bleCharacteristic) return;

  float _tiDist = radarCurrentDistanceM();
  float rdistM = fusedDist(_tiDist);
  // Stream the same EMA-smoothed HR/BR the radar screen shows so the
  // companion app graph doesn't jitter while the on-device text stays calm.
  float br     = fusedBr(isfinite(radarBreathBpmDisplay) ?
radarBreathBpmDisplay : radarBreathBpm);
  float hr     = fusedHr(isfinite(radarHeartBpmDisplay)  ?
radarHeartBpmDisplay  : radarHeartBpm);

  char rdistStr[12], brStr[10], hrStr[10], ussStr[12];
  if (isfinite(rdistM))    snprintf(rdistStr, sizeof(rdistStr), "%.2f", rdistM);
  else                     snprintf(rdistStr, sizeof(rdistStr), "nan");
  if (isfinite(br))        snprintf(brStr,    sizeof(brStr),    "%.1f", br);
  else                     snprintf(brStr,    sizeof(brStr),    "nan");
  if (isfinite(hr))        snprintf(hrStr,    sizeof(hrStr),    "%.1f", hr);
  else                     snprintf(hrStr,    sizeof(hrStr),    "nan");
  if (proxFiltered > 0.0f) snprintf(ussStr,   sizeof(ussStr),   "%.1f", proxFiltered);
  else                     snprintf(ussStr,   sizeof(ussStr),   "nan");
```

```
    char buf[80];
    snprintf(buf, sizeof(buf), "RADAR|%.1f|%s|%d|%s|%s|%s|%d",
        yawDeg, rdistStr, radarHuman ? 1 : 0,
        brStr, hrStr, ussStr, proxTooCloseLatched ? 1 : 0);

    bleCharacteristic->setValue((uint8_t*)buf, strlen(buf));
    for (uint16_t handle : bleConnHandles) {
        bleCharacteristic->notify(handle);
    }
}

// ── INS mode BLE packets ──────────────────────────────────────────────
//
// Two packets sent on bleCharacteristic (UTF-8 string notify) per update cycle
// at ~2 Hz while in INS_MODE with BLE connected.
//
void bleSendInsData() {
    if (bleState != BLE_CONNECTED || !bleCharacteristic) return;

    // ── Packet 1: telemetry ──────────────────────────────────────────
    float crx = startSet ? (displayX_m - startX) : 0.0f;
    float cry = startSet ? (displayY_m - startY) : 0.0f;
    float altDisplay = (isfinite(altitudeM) && isfinite(altitudeZeroM))
        ? (altitudeM - altitudeZeroM) : altitudeM;

    float    uwbSnap;
    uint32_t uwbMsSnap;
    bool     uwbValidSnap;
    float    uwbLastGoodSnap2;
    noInterrupts();
    uwbSnap        = uwbLatestDistanceM;
    uwbMsSnap      = uwbLatestUpdateMs;
    uwbValidSnap   = uwbLatestValid;
    uwbLastGoodSnap2 = uwbLastGoodDistanceM;
    interrupts();
    bool uwbFresh = uwbValidSnap && isfinite(uwbSnap) &&
                    ((millis() - uwbMsSnap) < UWB_DISPLAY_HOLD_MS);
    if (!uwbFresh && isfinite(uwbLastGoodSnap2)) { uwbSnap =
uwbLastGoodSnap2; }
    bool uwbHas = isfinite(uwbSnap);

    // Last victim snapshot
    int    vgps  = 0;
```

```
    double vlat  = 0.0, vlon = 0.0;
    float  ve_f  = 0.0f, vn_f = 0.0f;
    bool   vHasGps = false, vHasUwb = false;
    if (victimLogCount > 0) {
      int vi = (victimLogHead - 1 + VICTIM_LOG_MAX) % VICTIM_LOG_MAX;
      VictimSnapshot& vs = victimLog[vi];
      vgps = vs.gpsFix ? 1 : 0;
      if (vs.gpsFix) { vlat = vs.lat; vlon = vs.lon; vHasGps = true; }
      else           { ve_f = vs.e_m; vn_f = vs.n_m; vHasUwb = true; }
    }

    char latStr[14], lonStr[14], altStr[10], fdStr[10], uwbStr[10];
    char vlStr[14], voStr[14], veStr[10], vnStr[10];

    if (gpsValidNow) { snprintf(latStr, sizeof(latStr), "%.5f", lastLat);
                       snprintf(lonStr, sizeof(lonStr), "%.5f", lastLon); }
    else             { snprintf(latStr, sizeof(latStr), "nan");
                       snprintf(lonStr, sizeof(lonStr), "nan"); }
    if (isfinite(altDisplay))     snprintf(altStr, sizeof(altStr), "%.2f", altDisplay);
    else                          snprintf(altStr, sizeof(altStr), "nan");
    if (isfinite(fusedDistanceM)) snprintf(fdStr,  sizeof(fdStr),  "%.2f",
fusedDistanceM);
    else                          snprintf(fdStr,  sizeof(fdStr),  "nan");
    if (uwbHas)                       snprintf(uwbStr, sizeof(uwbStr), "%.2f", uwbSnap);
    else                          snprintf(uwbStr, sizeof(uwbStr), "nan");

    if (vHasGps)  { snprintf(vlStr, sizeof(vlStr), "%.5f", vlat);
                    snprintf(voStr, sizeof(voStr), "%.5f", vlon);
                    snprintf(veStr, sizeof(veStr), "nan");
                    snprintf(vnStr, sizeof(vnStr), "nan"); }
    else if (vHasUwb) { snprintf(vlStr, sizeof(vlStr), "nan");
                        snprintf(voStr, sizeof(voStr), "nan");
                        snprintf(veStr, sizeof(veStr), "%.2f", ve_f);
                        snprintf(vnStr, sizeof(vnStr), "%.2f", vn_f); }
    else          { snprintf(vlStr, sizeof(vlStr), "nan");
                    snprintf(voStr, sizeof(voStr), "nan");
                    snprintf(veStr, sizeof(veStr), "nan");
                    snprintf(vnStr, sizeof(vnStr), "nan"); }

    char t[220];
    snprintf(t, sizeof(t),

"INST|%d|%s|%s|%.2f|%.2f|%s|%s|%s|%.1f|%.1f|%.1f|%.2f|%.2f|%d|%d|%s|%s|
%s|%s",
        gpsValidNow ? 1 : 0,
        latStr, lonStr,
```

```
        crx, cry, altStr, fdStr, uwbStr,
        yawDegDisp, rollDegNowAbs, pitchDegNowAbs,
        vE, vN,
        victimCount, vgps,
        vlStr, voStr, veStr, vnStr);

    bleCharacteristic->setValue((uint8_t*)t, strlen(t));
    for (uint16_t handle : bleConnHandles) {
        bleCharacteristic->notify(handle);
    }

    // ── Packet 2: map grid
──────────────────────────────────────────────────────────────────────
─
    // Buffer: 5 prefix + 14 rows * (41 chars + 1 tilde) + 1 null = 594 bytes max.
    static char mapBuf[MAP_H * (MAP_W + 2) + 8];
    int pos = 0;
    mapBuf[pos++] = 'I'; mapBuf[pos++] = 'N'; mapBuf[pos++] = 'S';
mapBuf[pos++] = 'M';
    mapBuf[pos++] = '|';
    for (int r = 0; r < MAP_H; r++) {
        if (r > 0) mapBuf[pos++] = '~';
        int len = (int)strlen(prevInsGrid[r]);
        while (len > 0 && prevInsGrid[r][len - 1] == ' ') len--;
        memcpy(mapBuf + pos, prevInsGrid[r], len);
        pos += len;
    }
    mapBuf[pos] = '\0';

    if (pos <= 509) {
        bleCharacteristic->setValue((uint8_t*)mapBuf, pos);
        for (uint16_t handle : bleConnHandles) {
            bleCharacteristic->notify(handle);
        }
    // After the INSM map packet, this third block should exist:

    // ── Packet 3: INSV (victim log)
──────────────────────────────────────────────────────
    if (victimLogCount > 0) {
        static char vBuf[400];
        int vPos = 0;
        vPos += snprintf(vBuf + vPos, sizeof(vBuf) - vPos, "INSV|%d",
victimLogCount);
        for (int i = 0; i < victimLogCount; i++) {
            int vi = (victimLogHead - victimLogCount + i + VICTIM_LOG_MAX) %
VICTIM_LOG_MAX;
```

```
            VictimSnapshot& vs = victimLog[vi];
            char vlat[12], vlon[12], ve[10], vn[10];
            if (vs.gpsFix) {
                snprintf(vlat, sizeof(vlat), "%.5f", vs.lat);
                snprintf(vlon, sizeof(vlon), "%.5f", vs.lon);
                snprintf(ve,   sizeof(ve),   "nan");
                snprintf(vn,   sizeof(vn),   "nan");
            } else {
                snprintf(vlat, sizeof(vlat), "nan");
                snprintf(vlon, sizeof(vlon), "nan");
                snprintf(ve,   sizeof(ve),   "%.2f", vs.e_m);
                snprintf(vn,   sizeof(vn),   "%.2f", vs.n_m);
            }
            vPos += snprintf(vBuf + vPos, sizeof(vBuf) - vPos,
                "|%d|%s|%s|%s|%s",
                vs.gpsFix ? 1 : 0,
                vlat, vlon, ve, vn);
            if (vPos >= (int)sizeof(vBuf) - 40) break;
        }
        if (vPos <= 509) {
            bleCharacteristic->setValue((uint8_t*)vBuf, vPos);
            for (uint16_t handle : bleConnHandles) {
                bleCharacteristic->notify(handle);
            }
        }
    }
}
}

// Toggle BLE advertising / connected state from the menu tile.
void toggleBleConnection() {
    if (bleState == BLE_DISCONNECTED) {
        NimBLEDevice::getAdvertising()->start();
        bleState = BLE_ADVERTISING;
        bleNeedsRedraw = true;
    } else if (bleState == BLE_ADVERTISING) {
        NimBLEDevice::getAdvertising()->stop();
        bleState = BLE_DISCONNECTED;
        bleNeedsRedraw = true;
    } else if (bleState == BLE_CONNECTED) {
        NimBLEDevice::getAdvertising()->stop();
        for (uint16_t handle : bleConnHandles) {
            bleServer->disconnect(handle);
        }
        bleConnHandles.clear();
        bleState = BLE_DISCONNECTED;
```

```
        bleNeedsRedraw = true;
    }
}
// ── SCAN mode BLE packets ──────────────────────────────────────────────────
//
// Two packets sent at ~2 Hz while in SCAN_MODE with BLE connected.
//
// ── Packet 1: SCAN1 (phase + radar + IR) ──────────────────────────────────
//   SCAN1|<state>|<progress>|<remaining_s>|<hr>|<br>|<radar_conf>|
//         <ir_conf>|<ir_cat>|<ir_hot>|<ir_max_f>
//
//   state      : int    — ScanState enum value (0=IDLE..6=ABORT)
//   progress   : int    — 0-1000 permille progress
//   remaining  : float  — seconds remaining in current phase
//   hr         : float  — heart rate BPM; "nan" if unavailable
//   br         : float  — breathing rate BPM; "nan" if unavailable
//   radar_conf : float  — radar confidence 0.0-1.0
//   ir_conf    : float  — IR confidence 0.0-1.0
//   ir_cat     : string — top IR category label
//   ir_hot     : int    — hot pixel count
//   ir_max_f   : float  — max target temp in Fahrenheit
//
// Example: "SCAN1|2|350|14.2|72.1|15.3|0.82|0.75|STANDING|12|98.6"
//
// ── Packet 2: SCAN2 (fusion + position) ───────────────────────────────────
//   SCAN2|<fuse_conf>|<decision>|<warning>|<gps>|<uwb>|<ins>|
//         <lat>|<lon>|<rel_x>|<rel_y>|<yaw>|<active>
//
//   fuse_conf : float  — overall fusion confidence 0.0-1.0
//   decision  : string — fusion decision text
//   warning   : string — warning text; empty string if none
//   gps       : 0/1    — GPS position valid
//   uwb       : 0/1    — UWB position valid
//   ins       : 0/1    — INS position valid
//   lat       : float  — latitude (5dp); "nan" if no GPS
//   lon       : float  — longitude (5dp); "nan" if no GPS
//   rel_x     : float  — relative X position (m)
//   rel_y     : float  — relative Y position (m)
//   yaw       : float  — heading (degrees)
//   active    : 0/1    — scan currently running
//
// Example: "SCAN2|0.87|HUMAN DETECTED||1|1|1|34.05223|-
118.24368|5.23|-2.14|45.2|1"
```

```
void bleSendScanData() {
   if (bleState != BLE_CONNECTED || !bleCharacteristic) return;

   // Phase remaining calculation
   uint32_t phaseBudgetMs = 0;
   switch (scanRt.state) {
      case ScanMode::SCAN_INIT:     phaseBudgetMs = scanCfg.initTimeoutMs;
break;
      case ScanMode::SCAN_RADAR:    phaseBudgetMs =
scanCfg.radarMaxDurationMs;   break;
      case ScanMode::SCAN_IR:       phaseBudgetMs =
scanCfg.irMaxDurationMs;      break;
      case ScanMode::SCAN_POSITION: phaseBudgetMs =
scanCfg.positionMaxDurationMs; break;
      case ScanMode::SCAN_FUSION:   phaseBudgetMs =
scanCfg.fusionHoldMs;         break;
      default: phaseBudgetMs = 0; break;
   }
   float remainingS = 0.0f;
   if (phaseBudgetMs > 0 && scanRt.stateStartMs > 0) {
      uint32_t elapsed = millis() - scanRt.stateStartMs;
      remainingS = max(0.0f, (float)(phaseBudgetMs -
min<uint32_t>(phaseBudgetMs, elapsed)) / 1000.0f);
   }

   // HR / BR
   char hrStr[10], brStr[10];
   if (isfinite(scanRt.radar.heartRateBpm)) snprintf(hrStr, sizeof(hrStr), "%.1f",
scanRt.radar.heartRateBpm);
   else snprintf(hrStr, sizeof(hrStr), "nan");
   if (isfinite(scanRt.radar.breathingBpm)) snprintf(brStr, sizeof(brStr), "%.1f",
scanRt.radar.breathingBpm);
   else snprintf(brStr, sizeof(brStr), "nan");

   // IR category — use dash if empty
   const char* irCat = (scanRt.ir.topCategory[0]) ? scanRt.ir.topCategory : "-";
   float irMaxF = cToF(scanRt.ir.maxTargetTemp);

   // ── Packet 1
──────────────────────────────────────────────────────────────────
──────
   char p1[160];
   snprintf(p1, sizeof(p1),
      "SCAN1|%d|%d|%.1f|%s|%s|%.2f|%.2f|%s|%d|%.1f",
      (int)scanRt.state,
      (int)scanRt.progressPermille,
```

```
        remainingS,
        hrStr, brStr,
        scanRt.radar.radarConfidence,
        scanRt.ir.confidenceScore,
        irCat,
        scanRt.ir.hotPixelCount,
        irMaxF);

    bleCharacteristic->setValue((uint8_t*)p1, strlen(p1));
    for (uint16_t handle : bleConnHandles) {
        bleCharacteristic->notify(handle);
    }

    // ── Packet 2 ──────────────────────────────────────────────────────────
    char latStr[14], lonStr[14];
    if (scanRt.position.gnssValid) {
        snprintf(latStr, sizeof(latStr), "%.5f", scanRt.position.lat);
        snprintf(lonStr, sizeof(lonStr), "%.5f", scanRt.position.lon);
    } else {
        snprintf(latStr, sizeof(latStr), "nan");
        snprintf(lonStr, sizeof(lonStr), "nan");
    }

    // Decision and warning — replace any pipe chars to avoid parse corruption
    char decision[32], warning[32];
    strncpy(decision, scanRt.fusion.decisionText[0] ? scanRt.fusion.decisionText :
"-", sizeof(decision) - 1);
    strncpy(warning,  scanRt.fusion.warning[0]      ? scanRt.fusion.warning      : "",
sizeof(warning)  - 1);
    decision[sizeof(decision) - 1] = '\0';
    warning[sizeof(warning)   - 1] = '\0';
    // Sanitize pipes
    for (char* c = decision; *c; c++) if (*c == '|') *c = '/';
    for (char* c = warning;  *c; c++) if (*c == '|') *c = '/';

    char p2[180];
    snprintf(p2, sizeof(p2),
        "SCAN2|%.2f|%s|%s|%d|%d|%d|%s|%s|%.2f|%.2f|%.1f|%d",
        scanDisplayConfidence(),
        decision,
        warning,
        scanRt.position.gnssValid ? 1 : 0,
        scanRt.position.uwbValid  ? 1 : 0,
        scanRt.position.insValid  ? 1 : 0,
```

```
        latStr, lonStr,
        scanRt.position.relX_m,
        scanRt.position.relY_m,
        scanRt.position.yawDeg,
        scanRt.active ? 1 : 0);

    bleCharacteristic->setValue((uint8_t*)p2, strlen(p2));
    for (uint16_t handle : bleConnHandles) {
        bleCharacteristic->notify(handle);
    }
}

// Called every loop() tick. Redraws the BLE tile if its state changed and the
// menu is currently visible.
void serviceBLE() {
    if (bleNeedsRedraw && currentMode == MENU && menuDrawn) {
        lcdBeginAccess();
        drawBleTile();
        lcdEndAccess();
        bleNeedsRedraw = false;
    return;
    }
    // Wait 1500 ms after connection: service discovery (~500 ms) + Android's
    // own 500 ms delay before it writes the CCCD + CCCD write round-trip.
    if (bleSendConnectedLog && bleState == BLE_CONNECTED &&
            (millis() - bleConnectedMs >= 1500)) {
        bleSendLog("ESP32 SAYS CONNECTED");
        bleSendConnectedLog = false;
        // Keep advertising for additional connections
        NimBLEDevice::getAdvertising()->start();
    }
    // Stream thermal frames at ~3 Hz while in THERMAL mode and connected.
    static uint32_t lastThermalBleMs = 0;
    if (bleState == BLE_CONNECTED && currentMode == THERMAL &&
thermalFrameValid &&
            (millis() - lastThermalBleMs >= 333)) {
        lastThermalBleMs = millis();
        bleSendThermalFrame();
    }
        // Stream radar data at ~5 Hz while in PROXIMITY mode and connected.
    static uint32_t lastRadarBleMs = 0;
    if (bleState == BLE_CONNECTED && currentMode == PROXIMITY &&
            (millis() - lastRadarBleMs >= 200)) {
        lastRadarBleMs = millis();
        bleSendRadarData();
    }
```

```
    // Stream INS data at ~2 Hz while in INS_MODE and connected.
    static uint32_t lastInsBleMs = 0;
    if (bleState == BLE_CONNECTED && currentMode == INS_MODE &&
            (millis() - lastInsBleMs >= 500)) {
        lastInsBleMs = millis();
        bleSendInsData();
    }
    // Stream scan data at ~2 Hz while in SCAN_MODE and connected.
    static uint32_t lastScanBleMs = 0;
    if (bleState == BLE_CONNECTED && currentMode == SCAN_MODE &&
        (millis() - lastScanBleMs >= 500)) {
     lastScanBleMs = millis();
     bleSendScanData();
}

}

// ----------------- SETUP -----------------
// setup() initializes buses, sensors, screen state, and buffers. loop() then
// acts as the scheduler that services input, sensors, and the active screen.
void setup() {
  Serial.begin(115200);
  setCpuFrequencyMhz(240);
  delay(100);

    spiBusMutex = xSemaphoreCreateMutex();
    Wire.begin(SDA_PIN, SCL_PIN);
    Wire.setClock(400000);
    delay(50);

    pinMode(UWB_SS, OUTPUT);
    digitalWrite(UWB_SS, HIGH);
    pinMode(TOUCH_IRQ, INPUT);

    //CAT9555
    catSetPinModeP0(TFT_RST_BIT, true);
    catSetPinModeP0(TOUCH_CS_BIT, true);
    catSetPinModeP0(USS_TRIG_BIT, true);
    catSetPinModeP0(USS_ECHO_BIT, false);
    catSetPinModeP1(BTN_LEFT_BIT, false);
    catSetPinModeP1(BTN_UP_BIT, false);
    catSetPinModeP1(BTN_RIGHT_BIT, false);
    catSetPinModeP1(BTN_DOWN_BIT, false);
    catSetPinModeP1(BTN_CENTER_BIT, false);
    catSetPinModeP1(LED1_BIT, true);          // set as output
```

```
  catDigitalWriteP1(LED1_BIT, false);       // active high — drive LOW to ensure
OFF at boot
  catDigitalWriteP0(TOUCH_CS_BIT, true);
  catDigitalWriteP0(USS_TRIG_BIT, false);
  delay(50);
  catDigitalWriteP0(TFT_RST_BIT, false);
  delay(20);
  catDigitalWriteP0(TFT_RST_BIT, true);
  delay(120);

  // First bring up TFT completely
  SPI.begin(TFT_SCLK, TFT_MISO, TFT_MOSI, TFT_CS);
  pinMode(TFT_CS, OUTPUT);
  digitalWrite(TFT_CS, HIGH);
  tft.begin();
  tft.setRotation(3);
  lcdBeginAccess(); tft.fillScreen(ILI9341_BLACK); lcdEndAccess();
  delay(50);

  // One-time SPI port setup only — reset is handled properly inside setupUWB()
  spiBegin(UWB_IRQ, -1);
  spiSelect(UWB_SS);
  uwbSpiPortReady = true;

  //I2S audio
  audioOut=new VoiceTunedI2S();
  audioOut->SetPinout(I2S_BCLK,I2S_LRCLK,I2S_DIN);
  audioOut->SetGain(1.0f);
  audioOut->gainRamp=1.0f;
  wav=new AudioGeneratorWAV();
  audioOut->begin();
  delay(50);

  //MLX90640 THERMAL
  uint16_t ee[832];
  if (MLX90640_DumpEE(MLX90640_ADDR, ee) == 0) {
    MLX90640_ExtractParameters(ee, &mlx90640);
    MLX90640_SetRefreshRate(MLX90640_ADDR, MLX_IDLE_REFRESH);
    mlxReady = true;
  } delay(50);

  // IMU/BAROMETER
  imuReady = mpu.setup(0x68);
  delay(50);
  bmpReady = bmp.begin(0x76) || bmp.begin(0x77);
  if (bmpReady) {
```

```
        bmp.setSampling(
            Adafruit_BMP280::MODE_NORMAL,
            Adafruit_BMP280::SAMPLING_X2,
            Adafruit_BMP280::SAMPLING_X16,
            Adafruit_BMP280::FILTER_X16,
            Adafruit_BMP280::STANDBY_MS_500
        );
    }
    delay(50);

    // FILTER/SCAN/RADAR
    filter.begin(200.0f);
    ScanMode::resetRuntime(scanRt);
    insPrevUs = micros();
    thermalActive = false;
    insSensorsActive = false;
    // Pre-configure radar once at boot, then drop the data UART so later tile
    // switches can wake it without requiring a full module reset/power cycle.
    // radarStart();
    ensureRadarActive(false);
    delay(50);
    setupBLE();
    drawMenu();
    applyPeripheralMode(MENU);
    lastMode = MENU;
    lastInputMs = millis();

    playWelcomeIfIdle();

    initBleRadar2();  // start NimBLE client — will scan and connect to
Radar2Anchor
}

void loop() {
    // Tight-pump: nothing else runs while the TOOCLOSE alert plays.
    // wav->loop() must be called as fast as possible to keep the I2S DMA fed.
    //
    // IMPORTANT: serviceTooClose() is called AFTER this block (same place as
    // before this whole TOO-CLOSE-on-scan-screen change set). On the very
    // first iteration it fires, the rest of loop() still runs once so that
    // runProximity()/runScan() get a chance to call updateProximityReading()
    // and keep proxTooCloseLatched fresh. If serviceTooClose() ran before the
    // tight-pump, the pump would steal the whole iteration, the flag would
    // never be refreshed while the clip played, and the clip would restart
    // forever because the pump exits on AUDIO_DONE with the flag still true.
    if (tooCloseAlertPlaying) {
```

```
        // Also break if flag cleared mid-clip so audio stops immediately.
        while ((audioState != AUDIO_DONE && audioState != AUDIO_IDLE) &&
proxTooCloseLatched) {
            serviceAudio();
            yield();
        }
        stopAudioNow();
        tooCloseAlertPlaying = false;
        if (!proxTooCloseLatched) {
            if (currentMode == PROXIMITY) {
                proxScreenInit = false;
                proxRadarRestartPrompt = true;  // guarantee "Area clear" shows on
reinit
            } else if (currentMode == SCAN_MODE) {
                // Invalidate every scan cache so the scan screen repaints
                // cleanly over the red TOO CLOSE fillScreen. This mirrors the
                // PROXIMITY exit above.
                scanScreenInit = false;
                scanThermalReviewInit = false;
                scanNavReviewInit = false;
                prevThermalValid = false;
                thermalHudInit = false;
                topBarDirty = true;
                for (int i = 0; i < 10; i++) prevScanLine[i][0] = '\0';
                prevScanPhaseLabel[0] = '\0';
                prevScanPhaseTimer[0] = '\0';
                prevScanProgressFill = 0;
            }
        }
        return;
    }

    serviceTooClose();
    serviceBleRadar2();
    serviceAudio();
    serviceAlert();
    serviceLED();
    serviceBLE();

    if (currentMode == INS_MODE || currentMode == SCAN_MODE ||
currentMode == PROXIMITY) updateINS();

    //NAV BUTTON
    // Handles physical button navigation.
    // In MENU mode, arrows move between tiles and CENTER enters the
selected tile.
```

```
    // In app modes, arrows move toolbar selection and CENTER activates it.

    static NavButton lastNavBtn = NAV_NONE;
    NavButton navBtn = readNavButton();

    if (navBtn != NAV_NONE && lastNavBtn == NAV_NONE) {
        lastInputMs = millis();
bool audioActive =
    (audioState == AUDIO_STARTING ||
     audioState == AUDIO_PLAYING ||
     audioState == AUDIO_FINISHING);

bool clickRecovery =
    currentClip == CLIP_CLICK &&
    audioActive &&
    (millis() - audioStartMs < 50);   // short click lockout only

bool audioBlocking =
    audioActive &&
    !scanUiShouldIgnoreAudioBlock() &&
    (
        currentClip == CLIP_WELCOME ||
        currentClip == CLIP_SCAN ||
        currentClip == CLIP_HUMAN ||
        currentClip == CLIP_TOOCLOSE ||
        clickRecovery
    );

        if (!audioBlocking) {
            if (screensaverOn) {
                exitScreensaver();
            } else {
                handleNavButton(navBtn);
            }
        }
    }

    lastNavBtn = navBtn;


    // TOUCH
    // Touchscreen input.
    // A tap is accepted only after a short stability check. While the screensaver
    // is active, the first touch wakes the screen instead of activating UI.
    int x, y;
    bool touched = getTouchPoint(x, y);
```

```
    if (touched) {
        lastInputMs = millis();
        if (screensaverOn) {
            exitScreensaver();
            touchDown = touched;
            return;
        }
        if ((currentMode == THERMAL || currentMode == THERMAL_LOADING) &&
y < 28) {
            stopAudioNow(); resetAudioState();
            ensureThermalActive(false); thermalHumanDetected=false;
presenceScore=0;
            toolbarSelection = TOOLBAR_CLOSE;
            currentMode = MENU;
            touchDown = touched;
            return;
        }
        if (currentMode == SCAN_LOADING && y < 28) {
            stopAudioNow(); resetAudioState();
            toolbarSelection = TOOLBAR_CLOSE;
            currentMode = MENU;
            touchDown = touched;
            return;
        }
    }
    if (!screensaverOn && currentMode == MENU && (millis() - lastInputMs >=
IDLE_TIMEOUT_MS)) {
        enterScreensaver();
        touchDown = touched;
        return;
    }
    if (screensaverOn) {
        touchDown = touched;
        return;
    }
    if (touched && !touchDown) {
    bool audioBlocking =
      (currentClip == CLIP_SCAN ||
      currentClip == CLIP_HUMAN ||
      currentClip == CLIP_TOOCLOSE) &&
     (audioState == AUDIO_STARTING ||
      audioState == AUDIO_PLAYING ||
      audioState == AUDIO_FINISHING) &&
     !scanUiShouldIgnoreAudioBlock();

        if (!audioBlocking) {
```

```
            int x2, y2;
            delay(8);
            bool t2 = getTouchPoint(x2, y2);
            if (t2 && abs(x2 - x) <= 12 && abs(y2 - y) <= 12) {
                handleTouchXY(x2, y2);
            }
        }
    }

    // Mode Transistion
    if (currentMode != lastMode) {
        applyPeripheralMode(currentMode);
        topBarDirty = true;

        if (currentMode == MENU) {
            stopAudioNow(); resetAudioState();
            menuDrawn = false;
            drawMenu();
        }  else if(currentMode==THERMAL_LOADING){
            bleSendLog("TILE: THERMAL");
            menuDrawn=false;
            toolbarSelection=TOOLBAR_CLOSE;
            drawThermalLoadingScreen();
            updateLoadingBar(0.0f);
            resetAlert();                // arm hd.h for this session
            thermalHumanDetected=false; presenceScore=0;
humanIndicatorShown=false;
            startClip(CLIP_SCAN);        // play loading sound once
        }  else if(currentMode==SCAN_LOADING){
            bleSendLog("TILE: SCAN");
            menuDrawn=false;
            toolbarSelection=TOOLBAR_CLOSE;
            drawScanLoadingScreen();
            updateLoadingBar(0.0f);
            resetAlert();
            startClip(CLIP_SCAN);
        } else if (currentMode == THERMAL) {
            drawThermalBase();
            resetAudioState();
        } else if (currentMode == PROXIMITY) {
            bleSendLog("TILE: RADAR");
            menuDrawn = false;
            toolbarSelection = TOOLBAR_CLOSE;
            resetProximityReading();
            resetAudioState();
            proxScreenInit = false;
```

```
            // Reset tooclose clip state on mode entry
            // (serviceTooClose uses a static — force it clean)
            stopAudioNow();
        } else if (currentMode == INS_MODE) {
            bleSendLog("TILE: INS");
            stopAudioNow(); resetAudioState();
            menuDrawn = false;
            toolbarSelection = TOOLBAR_CLOSE;
            drawInsBase();
            for (int r = 0; r < MAP_H; r++) prevInsGrid[r][0] = '\0';
            for (int i = 0; i < 13; i++) prevInsLine[i][0] = '\0';
            insCacheInit = true;
            insScreenInit = false;
        } else if (currentMode == SCAN_MODE) {
            menuDrawn = false;
            toolbarSelection = TOOLBAR_CLOSE;
            scanScreenInit = false;
            scanReviewSequenceActive = false;
            scanReviewScreen = SCAN_REVIEW_SUMMARY;
            scanReviewCompletionLatched = false;
            scanThermalReviewInit = false;
            scanNavReviewInit = false;
            resetProximityReading();
            resetAudioState();
        }
        lastMode = currentMode;
    }

      // ================== LOADING SCREENS (THERMAL / SCAN)
==================
    if(currentMode==THERMAL_LOADING){
    if(audioState==AUDIO_STARTING)       updateLoadingBar(0.02f);
    else if(audioState==AUDIO_PLAYING)
updateLoadingBar(0.05f+0.85f*((float)fadeSample/(float)FADE_SAMPLES));
    else if(audioState==AUDIO_FINISHING) updateLoadingBar(0.95f);
    else if(audioState==AUDIO_DONE){
      updateLoadingBar(1.0f);
       delay(120);
      currentMode = THERMAL;
      }
    return;
    }
  if(currentMode==SCAN_LOADING){
    if(audioState==AUDIO_STARTING)       updateLoadingBar(0.02f);
    else if(audioState==AUDIO_PLAYING)
updateLoadingBar(0.05f+0.85f*((float)fadeSample/(float)FADE_SAMPLES));
```

```
    else if(audioState==AUDIO_FINISHING) updateLoadingBar(0.95f);
    else if(audioState==AUDIO_DONE){
      updateLoadingBar(1.0f);
      delay(120);
      currentMode = SCAN_MODE;
      }
    return;
    }
        if (currentMode == THERMAL) {
            runThermal();
            radarPoll();
        }
            else if (currentMode == PROXIMITY) {
            updateThermalDetection();
            runProximity();
        } else if (currentMode == INS_MODE) {
            if (uwbStartPending) {
            updateFusedDistance();
            runINS();
            ensureUwbActive(true);
            } else {
            pollUWB();
            updateFusedDistance();
            runINS();
            }
        } else if (currentMode == SCAN_MODE) {
            if (uwbStartPending) {
            runScan();
            ensureUwbActive(true);
            } else {
            runScan();
    }
      }
    }
```

3. Waveshare Radar2BLE header

```
#pragma once
/*
 * Radar2BLEClient.h
 * NimBLE client that connects to the Radar2Anchor BLE server and receives
 * breathing BPM, heart rate, and distance via notifications.
 *
 * Usage:
 * initBleRadar2() — call once in setup()
```

```
 * serviceBleRadar2() — call every loop(); non-blocking
 *
 * After connection, r2bpm / r2hr / r2distM update asynchronously via
 * notification callback. r2valid goes false when data is stale (>3 s).
 *
 * Fusion helpers (fusedBr, fusedHr, fusedDist) average TI radar + Radar2
 * when both are fresh, otherwise fall back to whichever is available.
 */
#include <NimBLEDevice.h>

// ── Must match the anchor's defines ──────────────────────────────────────
#define RADAR2_SERVICE_UUID "12345678-1234-1234-1234-123456789abc"
#define RADAR2_CHARACTERISTIC_UUID "abcdefab-cdef-abcd-efab-cdefabcdefab"

// Payload byte offsets (same as Radar2VitalsMsg.h on anchor side)
#define R2_FLAGS_IDX 0
#define R2_BPM_IDX 1
#define R2_HR_IDX 2
#define R2_DIST_IDX 3
#define R2_FLAGS_BPM 0x01
#define R2_FLAGS_HR 0x02
#define R2_FLAGS_DIST 0x04
#define R2_STALE_MS 3000UL // treat data stale after 3 s with no notification
#define R2_RETRY_MS 10000UL // retry scan every 10 s if not connected

// ── Live Radar2 values (updated by notification callback) ────────────────
static float r2bpm = NAN;
static float r2hr = NAN;
static float r2distM = NAN;
static bool r2valid = false;
static unsigned long r2lastMs = 0;

// ── Internal state ───────────────────────────────────────────────────────
static NimBLEClient* _r2Client = nullptr;
static bool _r2Connected = false;
```

```
47. static bool _r2DevFound = false;
48. static NimBLEAddress _r2FoundAddr("", BLE_ADDR_PUBLIC);
49.
50. // ── Notification callback ──────────────────────────────────────────────
51. static void _r2NotifyCb(NimBLERemoteCharacteristic*, uint8_t* data, size_t len, bool) {
52.     if (len < 1) return;
53.     const uint8_t flags = data[R2_FLAGS_IDX];
54.     r2bpm = (flags & R2_FLAGS_BPM) && len > R2_BPM_IDX ? (float)data[R2_BPM_IDX] : NAN;
55.     r2hr  = (flags & R2_FLAGS_HR)  && len > R2_HR_IDX  ? (float)data[R2_HR_IDX]  : NAN;
56.     r2distM = NAN;
57.     if ((flags & R2_FLAGS_DIST) && len >= R2_DIST_IDX + 3) {
58.         uint32_t cm = (uint32_t)data[R2_DIST_IDX]
59.                     | ((uint32_t)data[R2_DIST_IDX + 1] << 8)
60.                     | ((uint32_t)data[R2_DIST_IDX + 2] << 16);
61.         r2distM = cm / 100.0f;
62.     }
63.     r2valid = (flags != 0);
64.     r2lastMs = millis();
65. }
66.
67. // ── Client callbacks ──────────────────────────────────────────────────
68. class _R2ClientCB : public NimBLEClientCallbacks {
69.     void onDisconnect(NimBLEClient* pClient, int reason) override {
70.         _r2Connected = false;
71.         r2valid = false;
72.         r2bpm = r2hr = r2distM = NAN;
73.         Serial.println("[Radar2BLE] disconnected — will rescan");
74.     }
75. };
76.
77. // ── Scan callback — stops scan as soon as anchor is found ─────────────
78. class _R2ScanCB : public NimBLEScanCallbacks {
79.     void onResult(const NimBLEAdvertisedDevice* div) override {
80.         if (div->isAdvertisingService(NimBLEUUID(RADAR2_SERVICE_UUID))) {
81.             _r2FoundAddr = div->getAddress();
```

```
            _r2DevFound = true;
            NimBLEDevice::getScan()->stop();
            Serial.println("[Radar2BLE] anchor found");
        }
    }
};

// ── Public API ──────────────────────────────────────────────────────────
inline void initBleRadar2() {
    NimBLEDevice::init("");
    NimBLEDevice::setPower(ESP_PWR_LVL_P9);
    NimBLEScan* pScan = NimBLEDevice::getScan();
    pScan->setScanCallbacks(new _R2ScanCB());
    pScan->setActiveScan(true);
    pScan->setInterval(100);
    pScan->setWindow(99);
    Serial.println("[Radar2BLE] init done");
}

inline void serviceBleRadar2() {
    unsigned long now = millis();

    // Mark stale if no notification received recently
    if (r2valid && (now - r2lastMs > R2_STALE_MS)) {
        r2valid = false;
        r2bpm = r2hr = r2distM = NAN;
    }

    if (_r2Connected) return; // notifications are async — nothing to poll

    // If scan found the anchor, connect now
    if (_r2DevFound) {
        _r2DevFound = false;
        if (_r2Client) {
            NimBLEDevice::deleteClient(_r2Client);
```

```
117.           _r2Client = nullptr;
118.       }
119.       _r2Client = NimBLEDevice::createClient();
120.       _r2Client->setClientCallbacks(new _R2ClientCB(), true);
121.       _r2Client->setConnectionParams(12, 12, 0, 51);
122.
123.       if (_r2Client->connect(_r2FoundAddr)) {
124.           NimBLERemoteService* svc = _r2Client->getService(RADAR2_SERVICE_UUID);
125.           if (svc) {
126.               NimBLERemoteCharacteristic* ch = svc-
    >getCharacteristic(RADAR2_CHARACTERISTIC_UUID);
127.               if (ch && ch->canNotify()) {
128.                   ch->subscribe(true, _r2NotifyCb);
129.                   _r2Connected = true;
130.                   Serial.println("[Radar2BLE] connected + subscribed");
131.                   return;
132.               }
133.           }
134.           _r2Client->disconnect();
135.           Serial.println("[Radar2BLE] service/char not found");
136.       } else {
137.           Serial.println("[Radar2BLE] connect failed");
138.       }
139.       return;
140.   }
141.
142.   // Start a fresh scan periodically
143.   NimBLEScan* pScan = NimBLEDevice::getScan();
144.   if (pScan->isScanning()) return;
145.
146.   static unsigned long lastScanMs = 0;
147.   if (now - lastScanMs < R2_RETRY_MS) return;
148.
149.   lastScanMs = now;
150.   pScan->clearResults();
151.   pScan->start(5, false); // 5 s background scan; _R2ScanCB fires on find
152.   Serial.println("[Radar2BLE] scanning...");
```

```
}

// ── Fusion helpers ──────────────────────────────────────────────────────────
// Average TI radar + Radar2 when both are fresh; fall back to whichever exists.
inline float fusedBr(float tiBr) {
    bool r2fresh = r2valid && isfinite(r2bpm);
    bool tiFresh = isfinite(tiBr);
    if (tiFresh && r2fresh) return (tiBr + r2bpm) * 0.5f;
    if (r2fresh) return r2bpm;
    return tiBr; // NAN if neither available
}

inline float fusedHr(float tiHr) {
    bool r2fresh = r2valid && isfinite(r2hr);
    bool tiFresh = isfinite(tiHr);
    if (tiFresh && r2fresh) return (tiHr + r2hr) * 0.5f;
    if (r2fresh) return r2hr;
    return tiHr;
}

inline float fusedDist(float tiDist) {
    bool r2fresh = r2valid && isfinite(r2distM);
    bool tiFresh = isfinite(tiDist);
    if (tiFresh && r2fresh) return (tiDist + r2distM) * 0.5f;
    if (r2fresh) return r2distM;
    return tiDist;
}
```

4. Extended Kalman Filter header

```
#ifndef HANDHELD_EKF_H
#define HANDHELD_EKF_H

// ============================================================================
// handheld_ekf.h — 8-state Extended Kalman Filter for handheld IMU/UWB/baro
//
// State vector x[8]:
//   [X, Y, Z]          — 3D position in meters (local ENU frame, origin at init)
```

```
//   [VX, VY, VZ]       — 3D velocity in m/s
//   [YAW]              — heading in radians, maintained as gyro-Z integral
//   [BARO_BIAS]        — barometric altitude offset in meters (random walk)
//
// Prediction: driven by MPU9250 IMU at ~100 Hz via predictFromImu().
// Updates:
//   - UWB ranging anchor via updateFromUwb() — corrects X, Y, Z independently
//   - BMP280 barometer via updateFromBaro() — corrects Z + BARO_BIAS jointly
//
// Roll and pitch are NOT part of the EKF state. They are maintained by a
// separate complementary filter (updateRollPitchComplementary) that is cheap
// and sufficient for the slow tilt dynamics of a handheld device. Yaw IS in
// the EKF state so that UWB position updates can implicitly correct heading
// error through the cross-covariance terms.
//
// All matrix operations are fixed 8×8 to avoid heap allocation on Arduino.
// ============================================================================

#include <Arduino.h>
#include <math.h>
#include <string.h>

namespace HandheldEkf {

// Number of EKF state dimensions — fixed at compile time for stack allocation.
static constexpr int EKF_N = 8;
// Standard gravity used for body-to-world acceleration transform.
static constexpr float GRAVITY_MPS2 = 9.80665f;

// ---------------------------------------------------------------------------
// StateIndex — named offsets into x[] and P[][] for readability.
// ---------------------------------------------------------------------------
enum StateIndex : uint8_t {
  IDX_X = 0,       // East position (m)
  IDX_Y,           // North position (m)
  IDX_Z,           // Up position (m)
  IDX_VX,          // East velocity (m/s)
```

```
  IDX_VY,          // North velocity (m/s)
  IDX_VZ,          // Up velocity (m/s)
  IDX_YAW,         // Heading (rad), maintained via gyro-Z integration
  IDX_BARO_BIAS    // Barometer altitude offset (m), modeled as random walk
};

// ---------------------------------------------------------------------------
// ImuSample — one reading from the MPU9250, in SI units.
// t_us is the Arduino micros() timestamp at sample time.
// ---------------------------------------------------------------------------
struct ImuSample {
  uint32_t t_us;    // Timestamp (µs)
  float ax_mps2;    // Accelerometer X (m/s²) — body frame
  float ay_mps2;    // Accelerometer Y (m/s²) — body frame
  float az_mps2;    // Accelerometer Z (m/s²) — body frame
  float gx_rps;     // Gyroscope X (rad/s)
  float gy_rps;     // Gyroscope Y (rad/s)
  float gz_rps;     // Gyroscope Z (rad/s) — used for yaw integration
};

// ---------------------------------------------------------------------------
// UwbSample — one ranging result from the DWM3000 anchor.
// has_x/y/z flags allow partial updates when only some axes are available
// (e.g., a 2D-only anchor configuration).
// std_xy_m / std_z_m are the 1-sigma ranging uncertainty passed into R.
// ---------------------------------------------------------------------------
struct UwbSample {
  uint32_t t_us;    // Timestamp (µs)
  bool has_x;       // True if x_m is valid
  bool has_y;       // True if y_m is valid
  bool has_z;       // True if z_m is valid
  float x_m;        // Measured X position (m)
  float y_m;        // Measured Y position (m)
  float z_m;        // Measured Z position (m)
  float std_xy_m;   // 1-sigma horizontal ranging uncertainty (m)
  float std_z_m;    // 1-sigma vertical ranging uncertainty (m)
};
```

```
// ---------------------------------------------------------------------------
// BaroSample — one altitude reading from the BMP280.
// altitude_m is the pressure-derived altitude; std_m is the measurement noise.
// The EKF simultaneously estimates altitude (IDX_Z) and the baro bias
// (IDX_BARO_BIAS), so the predicted measurement is Z + BARO_BIAS.
// ---------------------------------------------------------------------------
struct BaroSample {
  uint32_t t_us;    // Timestamp (µs)
  float altitude_m; // Barometric altitude (m)
  float std_m;      // 1-sigma measurement noise (m)
};

// ---------------------------------------------------------------------------
// PoseEstimate — caller-facing output from getPoseEstimate().
// pos_confidence and vel_confidence are 1/(1+variance) — closer to 1 means
// the EKF is certain; approaches 0 as diagonal covariance grows large.
// ---------------------------------------------------------------------------
struct PoseEstimate {
  float x_m;          // East position (m)
  float y_m;          // North position (m)
  float z_m;          // Up position (m)
  float vx_mps;       // East velocity (m/s)
  float vy_mps;       // North velocity (m/s)
  float vz_mps;       // Up velocity (m/s)
  float roll_rad;     // Roll from complementary filter (rad)
  float pitch_rad;    // Pitch from complementary filter (rad)
  float yaw_rad;      // Yaw from EKF state (rad)
  float baro_bias_m;  // Estimated barometer offset (m)
  float pos_confidence; // 1/(1 + mean_pos_variance), range [0,1]
  float vel_confidence; // 1/(1 + mean_vel_variance), range [0,1]
};

// ---------------------------------------------------------------------------
// SensorHealth — diagnostic flags updated every predict/update cycle.
// Callers can use these to degrade fusion weights or warn the user.
// ---------------------------------------------------------------------------
```

```
struct SensorHealth {
  bool imu_ok;          // True if predictFromImu() ran this cycle
  bool uwb_ok;          // True if last UWB update was accepted (passed gate)
  bool baro_ok;         // True if baro is updating and not timed out
  bool uwb_timeout;     // True if no UWB sample received within uwb_timeout_us
  bool baro_bias_large; // True if |BARO_BIAS| > baro_bias_large_threshold_m
};

// ---------------------------------------------------------------------------
// EkfConfig — all tunable parameters. Use defaultConfig() as a starting point.
// ---------------------------------------------------------------------------
struct EkfConfig {
  // ── Process noise (Q diagonal) ──────────────────────────────────────────
  // These represent uncertainty injected each prediction step. Larger values
  // make the filter trust measurements more and track faster, at the cost of
  // noisier estimates. Units are in the natural units of each state.
  float accel_noise_mps2;    // IMU accelerometer noise floor (m/s²) — added to vel Q
  float gyro_noise_rps;      // IMU gyro noise floor (rad/s) — added to yaw Q
  float yaw_process_noise;   // Extra yaw uncertainty per step (rad) — accounts for heading drift
  float baro_bias_walk_mps;  // Baro bias random walk rate (m/s) — slow drift model
  float pos_process_noise;   // Position uncertainty growth per step (m) — dead reckoning drift
  float vel_process_noise;   // Velocity uncertainty growth per step (m/s)

  // ── Roll/pitch complementary filter ─────────────────────────────────────
  // alpha=0.03 means the accelerometer corrects ~3% of the gyro estimate
  // each step — slow enough to reject vibration, fast enough to track tilt.
  float roll_pitch_alpha;    // Accel correction weight [0,1]; default 0.03
  float max_tilt_rad;        // Hard clamp on roll/pitch to prevent gimbal singularities
  float max_dt_s;            // Maximum allowed dt; longer gaps are clamped to prevent instability

  // ── Innovation gating ───────────────────────────────────────────────────
  // Reject measurements whose normalized innovation exceeds gateSigma.
  // 3.5σ UWB gate rejects ~0.02% of good readings and most multipath outliers.
  // 3.0σ baro gate is tighter because baro noise is more Gaussian.
  float uwb_gate_sigma;      // UWB chi-squared gate threshold (σ), default 3.5
  float baro_gate_sigma;     // Baro chi-squared gate threshold (σ), default 3.0
```

```
  // ── Sensor timeout thresholds ──────────────────────────────────────────
  uint32_t uwb_timeout_us;   // Mark UWB timed-out after this many µs with no sample
  uint32_t baro_timeout_us;  // Mark baro timed-out after this many µs with no sample

  // ── Initial covariance (P diagonal) ────────────────────────────────────
  // High initial variance reflects uncertainty about starting state.
  // UWB updates will rapidly converge position once ranging begins.
  float initial_pos_var;         // Initial position variance (m²)
  float initial_vel_var;         // Initial velocity variance (m²/s²)
  float initial_yaw_var;         // Initial yaw variance (rad²)
  float initial_baro_bias_var;   // Initial baro bias variance (m²)

  // ── Health diagnostics ─────────────────────────────────────────────────
  float baro_bias_large_threshold_m; // Flag baro_bias_large if |bias| exceeds this (m)
};

// ---------------------------------------------------------------------------
// EkfState — full mutable EKF state. Caller allocates one per filter instance.
// ---------------------------------------------------------------------------
struct EkfState {
  float x[EKF_N];       // State vector: [X, Y, Z, VX, VY, VZ, YAW, BARO_BIAS]
  float P[EKF_N][EKF_N]; // State covariance — symmetric positive definite 8×8 matrix

  // Roll and pitch maintained outside the EKF state by the complementary filter.
  // They feed bodyAccelToWorld() each prediction step but are not updated by UWB/baro.
  float roll_rad;
  float pitch_rad;

  // Timestamps of the most recent input of each type (µs) — used for dt and timeout logic.
  uint32_t last_predict_us;
  uint32_t last_uwb_us;
  uint32_t last_baro_us;

  SensorHealth health;
};
```

```
// ——— Utility functions ——————————————————————————————————————————

// Wrap angle to [-π, +π] — used after yaw integration to prevent unbounded growth.
static inline float wrapPi(float a) {
  while (a > PI) a -= 2.0f * PI;
  while (a < -PI) a += 2.0f * PI;
  return a;
}

// Clamp float to [lo, hi].
static inline float clampf(float v, float lo, float hi) {
  return (v < lo) ? lo : ((v > hi) ? hi : v);
}

// ——— Fixed 8×8 matrix operations (no heap, no dynamic allocation) ——————————

// Zero all elements of an 8×8 matrix.
static inline void zeroMat(float M[EKF_N][EKF_N]) {
  memset(M, 0, sizeof(float) * EKF_N * EKF_N);
}

// Set M to a scaled identity matrix (diag × I).
static inline void eyeMat(float M[EKF_N][EKF_N], float diag = 1.0f) {
  zeroMat(M);
  for (int i = 0; i < EKF_N; i++) M[i][i] = diag;
}

// Copy A into B.
static inline void copyMat(const float A[EKF_N][EKF_N], float B[EKF_N][EKF_N]) {
  memcpy(B, A, sizeof(float) * EKF_N * EKF_N);
}

// C = A × B — uses a temporary to allow in-place calls where C aliases A or B.
// Inner loop walks k first to maximize cache locality on A's rows.
static inline void matMul8(const float A[EKF_N][EKF_N], const float B[EKF_N][EKF_N], float
C[EKF_N][EKF_N]) {
```

```
  float tmp[EKF_N][EKF_N];
  zeroMat(tmp);
  for (int i = 0; i < EKF_N; i++) {
    for (int k = 0; k < EKF_N; k++) {
      float a = A[i][k];
      for (int j = 0; j < EKF_N; j++) {
        tmp[i][j] += a * B[k][j];
      }
    }
  }
  copyMat(tmp, C);
}

// At[j][i] = A[i][j] — used to compute F^T in the covariance propagation P = FPF^T + Q.
static inline void transpose8(const float A[EKF_N][EKF_N], float At[EKF_N][EKF_N]) {
  for (int i = 0; i < EKF_N; i++) {
    for (int j = 0; j < EKF_N; j++) At[j][i] = A[i][j];
  }
}

// A += B (in-place element-wise add) — used to add Q to FPF^T.
static inline void matAdd8(float A[EKF_N][EKF_N], const float B[EKF_N][EKF_N]) {
  for (int i = 0; i < EKF_N; i++) {
    for (int j = 0; j < EKF_N; j++) A[i][j] += B[i][j];
  }
}

// ---------------------------------------------------------------------------
// defaultConfig — tuned for handheld use with MPU9250 + BMP280 + DWM3000.
// accel_noise 1.8 m/s² reflects handheld motion; vel_process 0.30 allows
// fast tracking of walking speed transitions. UWB gate 3.5σ is permissive
// enough to keep good multipath-distorted readings while rejecting gross outliers.
// ---------------------------------------------------------------------------
static inline EkfConfig defaultConfig() {
  EkfConfig c{};
  c.accel_noise_mps2 = 1.8f;
  c.gyro_noise_rps = 0.08f;
```

```
  c.yaw_process_noise = 0.03f;
  c.baro_bias_walk_mps = 0.005f;
  c.pos_process_noise = 0.05f;
  c.vel_process_noise = 0.30f;

  c.roll_pitch_alpha = 0.03f;  // Very low — mostly trusts gyro, slowly corrects toward accel
  c.max_tilt_rad = 1.2f;       // ~69° — avoids gimbal lock near vertical
  c.max_dt_s = 0.05f;          // Cap dt at 50 ms; longer gaps indicate missed samples

  c.uwb_gate_sigma = 3.5f;     // Reject UWB innovations > 3.5σ (~0.02% of Gaussian readings)
  c.baro_gate_sigma = 3.0f;

  c.uwb_timeout_us = 1500000UL;  // 1.5 s
  c.baro_timeout_us = 1500000UL; // 1.5 s

  c.initial_pos_var = 1.0f;          // 1 m² — high initial uncertainty
  c.initial_vel_var = 1.0f;
  c.initial_yaw_var = 0.5f;
  c.initial_baro_bias_var = 0.5f;

  c.baro_bias_large_threshold_m = 3.0f;
  return c;
}

// ---------------------------------------------------------------------------
// initEkf — zero all state and set the initial state and covariance.
// UWB and baro start as unhealthy until their first update is accepted.
// ---------------------------------------------------------------------------
static inline void initEkf(EkfState& ekf, const EkfConfig& cfg, float x0 = 0.0f, float y0 = 0.0f, float z0 = 0.0f,
float yaw0 = 0.0f) {
  memset(&ekf, 0, sizeof(ekf));
  ekf.x[IDX_X] = x0;
  ekf.x[IDX_Y] = y0;
  ekf.x[IDX_Z] = z0;
  ekf.x[IDX_YAW] = yaw0;
  // Velocities, baro bias, roll, pitch all start at zero.
```

```
  // Diagonal P — off-diagonals are zero, reflecting no initial cross-correlations.
  zeroMat(ekf.P);
  ekf.P[IDX_X][IDX_X] = cfg.initial_pos_var;
  ekf.P[IDX_Y][IDX_Y] = cfg.initial_pos_var;
  ekf.P[IDX_Z][IDX_Z] = cfg.initial_pos_var;
  ekf.P[IDX_VX][IDX_VX] = cfg.initial_vel_var;
  ekf.P[IDX_VY][IDX_VY] = cfg.initial_vel_var;
  ekf.P[IDX_VZ][IDX_VZ] = cfg.initial_vel_var;
  ekf.P[IDX_YAW][IDX_YAW] = cfg.initial_yaw_var;
  ekf.P[IDX_BARO_BIAS][IDX_BARO_BIAS] = cfg.initial_baro_bias_var;

  ekf.health.imu_ok = true;
  ekf.health.uwb_ok = false;
  ekf.health.baro_ok = false;
  ekf.health.uwb_timeout = true;   // Assume no UWB until proven otherwise
  ekf.health.baro_bias_large = false;
}

// Full reset — thin wrapper around initEkf with explicit starting pose.
static inline void resetEkf(EkfState& ekf, const EkfConfig& cfg, float x0, float y0, float z0, float yaw0) {
  initEkf(ekf, cfg, x0, y0, z0, yaw0);
}

// ---------------------------------------------------------------------
// updateRollPitchComplementary — complementary filter for roll and pitch.
//
// Step 1: Derive roll and pitch from the accelerometer gravity vector.
//   acc_roll  = atan2(ay, az)             — tilt around X axis
//   acc_pitch = atan2(-ax, √(ay²+az²))   — tilt around Y axis
//   These are accurate when stationary but noisy during motion (linear
//   acceleration contaminates the gravity vector).
//
// Step 2: Integrate gyroscope to propagate the previous angle estimate.
//   roll  += gx * dt
//   pitch += gy * dt
//   Accurate during motion but drifts over time (gyro bias).
//
```

```
// Step 3: Fuse with EMA weighted toward gyro (1 - alpha = 0.97 gyro weight).
//   roll  = 0.97 * gyro_roll  + 0.03 * acc_roll
//   pitch = 0.97 * gyro_pitch + 0.03 * acc_pitch
//   Low alpha prevents accelerometer vibration from injecting noise while
//   still correcting gyro drift on the timescale of ~33 samples.
//
// Step 4: Clamp to max_tilt_rad (default 1.2 rad ≈ 69°) to avoid near-vertical
//   singularities where atan2 becomes unreliable.
// ---------------------------------------------------------------------------
static inline void updateRollPitchComplementary(EkfState& ekf, const ImuSample& imu, float dt, const
EkfConfig& cfg) {
  // Accelerometer-derived tilt angles (valid when gravity dominates)
  float acc_roll  = atan2f(imu.ay_mps2, imu.az_mps2);
  float acc_pitch = atan2f(-imu.ax_mps2, sqrtf(imu.ay_mps2 * imu.ay_mps2 + imu.az_mps2 * imu.az_mps2));

  // Gyroscope integration — propagate previous estimate forward
  ekf.roll_rad  += imu.gx_rps * dt;
  ekf.pitch_rad += imu.gy_rps * dt;

  // EMA correction toward accelerometer — (1-alpha) * gyro + alpha * accel
  ekf.roll_rad  = (1.0f - cfg.roll_pitch_alpha) * ekf.roll_rad  + cfg.roll_pitch_alpha * acc_roll;
  ekf.pitch_rad = (1.0f - cfg.roll_pitch_alpha) * ekf.pitch_rad + cfg.roll_pitch_alpha * acc_pitch;

  // Hard clamp to prevent singularity in the DCM at near-vertical orientation
  ekf.roll_rad  = clampf(ekf.roll_rad,  -cfg.max_tilt_rad, cfg.max_tilt_rad);
  ekf.pitch_rad = clampf(ekf.pitch_rad, -cfg.max_tilt_rad, cfg.max_tilt_rad);
}

// ---------------------------------------------------------------------------
// bodyAccelToWorld — rotate body-frame acceleration to world frame and
// subtract gravity, yielding the linear acceleration in ENU coordinates.
//
// Uses the full ZYX (yaw-pitch-roll) direction cosine matrix (DCM):
//   R = Rz(yaw) · Ry(pitch) · Rx(roll)
//
// The 9 DCM elements expanded:
//   R11 = cy·cp       R12 = cy·sp·sr - sy·cr    R13 = cy·sp·cr + sy·sr
```

```
//   R21 = sy·cp       R22 = sy·sp·sr + cy·cr    R23 = sy·sp·cr - cy·sr
//   R31 = -sp         R32 = cp·sr               R33 = cp·cr
//
// After rotation, gravity (9.80665 m/s² downward = -Z in world frame)
// is subtracted from az_world so the output is pure linear acceleration.
// If the device is stationary and level, ax_w ≈ ay_w ≈ az_w ≈ 0.
// ----------------------------------------------------------------------------
static inline void bodyAccelToWorld(
  float roll, float pitch, float yaw,
  float ax_b, float ay_b, float az_b,
  float& ax_w, float& ay_w, float& az_w
) {
  float cr = cosf(roll),  sr = sinf(roll);
  float cp = cosf(pitch), sp = sinf(pitch);
  float cy = cosf(yaw),   sy = sinf(yaw);

  // ZYX DCM rows
  float R11 = cy * cp;
  float R12 = cy * sp * sr - sy * cr;
  float R13 = cy * sp * cr + sy * sr;
  float R21 = sy * cp;
  float R22 = sy * sp * sr + cy * cr;
  float R23 = sy * sp * cr - cy * sr;
  float R31 = -sp;
  float R32 = cp * sr;
  float R33 = cp * cr;

  // Rotate body accel to world frame, then remove gravity from Z
  ax_w = R11 * ax_b + R12 * ay_b + R13 * az_b;
  ay_w = R21 * ax_b + R22 * ay_b + R23 * az_b;
  az_w = R31 * ax_b + R32 * ay_b + R33 * az_b - GRAVITY_MPS2;
}

// ----------------------------------------------------------------------------
// predictFromImu — EKF predict step driven by one IMU sample.
//
// Called at ~100 Hz. Performs:
```

```
//   1. Compute dt from timestamp delta; clamp to [0.5 ms, max_dt_s].
//   2. Update roll/pitch via complementary filter.
//   3. Rotate IMU accel to world frame and subtract gravity.
//   4. Propagate state with constant-acceleration kinematics:
//        pos += vel*dt + 0.5*a*dt²
//        vel += a*dt
//        yaw += gz*dt  (wrapped to [-π, +π])
//   5. Build linearized state transition matrix F (identity + velocity coupling).
//   6. Build process noise matrix Q (diagonal, dt-scaled).
//   7. Propagate covariance: P = F·P·Fᵀ + Q
//
// F is nearly identity — the only off-diagonal non-zero entries are:
//   F[IDX_X][IDX_VX] = dt,  F[IDX_Y][IDX_VY] = dt,  F[IDX_Z][IDX_VZ] = dt
// These capture how velocity uncertainty propagates into position uncertainty.
//
// Q is diagonal. Velocity Q combines the configured vel_process_noise and the
// IMU accelerometer noise floor (both squared then added — independent noise
// sources add in quadrature). Position Q scales with dt² because positional
// uncertainty comes from integrating velocity uncertainty. Yaw Q adds gyro
// noise to prevent the heading covariance from collapsing to zero.
// ---------------------------------------------------------------------
inline void predictFromImu(EkfState& ekf, const EkfConfig& cfg, const ImuSample& imu) {
  // Bootstrap: record timestamp on first call and return without predicting.
  if (ekf.last_predict_us == 0) {
    ekf.last_predict_us = imu.t_us;
    return;
  }

  float dt = (imu.t_us - ekf.last_predict_us) * 1e-6f;
  ekf.last_predict_us = imu.t_us;
  // Clamp dt: lower bound prevents division instability; upper bound caps
  // integration error from missed IMU samples (e.g. during BLE activity).
  dt = clampf(dt, 0.0005f, cfg.max_dt_s);

  // Step 1: Update attitude via complementary filter (roll/pitch only).
  updateRollPitchComplementary(ekf, imu, dt, cfg);
```

```
// Step 2: Rotate body accel to world frame, remove gravity.
float ax_w, ay_w, az_w;
bodyAccelToWorld(ekf.roll_rad, ekf.pitch_rad, ekf.x[IDX_YAW],
                 imu.ax_mps2, imu.ay_mps2, imu.az_mps2,
                 ax_w, ay_w, az_w);

// Step 3: Constant-acceleration kinematics — second-order Euler integration.
// The 0.5*a*dt² term improves accuracy over first-order at low IMU rates.
ekf.x[IDX_X]  += ekf.x[IDX_VX] * dt + 0.5f * ax_w * dt * dt;
ekf.x[IDX_Y]  += ekf.x[IDX_VY] * dt + 0.5f * ay_w * dt * dt;
ekf.x[IDX_Z]  += ekf.x[IDX_VZ] * dt + 0.5f * az_w * dt * dt;
ekf.x[IDX_VX] += ax_w * dt;
ekf.x[IDX_VY] += ay_w * dt;
ekf.x[IDX_VZ] += az_w * dt;
// Yaw integrated from gyro Z — not Madgwick, not accelerometer-corrected.
// This is intentional: Madgwick heading drifts in magnetically noisy environments.
ekf.x[IDX_YAW] = wrapPi(ekf.x[IDX_YAW] + imu.gz_rps * dt);
// BARO_BIAS is not propagated — it evolves only through Q (random walk model).

// Step 4: Build linearized state transition matrix F.
// F ≈ I because all state derivatives are simple: vel → pos coupling captured
// by the off-diagonal dt entries. Acceleration coupling (vel → vel via a) is
// absorbed into Q rather than F to keep the matrix sparse.
float F[EKF_N][EKF_N];
eyeMat(F, 1.0f);
F[IDX_X][IDX_VX] = dt;   // position updates with velocity
F[IDX_Y][IDX_VY] = dt;
F[IDX_Z][IDX_VZ] = dt;

// Step 5: Build process noise matrix Q.
// Squaring converts std-dev parameters to variance before scaling by dt.
float q_pos  = cfg.pos_process_noise * cfg.pos_process_noise;
float q_vel  = cfg.vel_process_noise * cfg.vel_process_noise;
float q_yaw  = cfg.yaw_process_noise * cfg.yaw_process_noise;
float q_baro = cfg.baro_bias_walk_mps * cfg.baro_bias_walk_mps;

float Q[EKF_N][EKF_N];
```

```
  zeroMat(Q);
  Q[IDX_X][IDX_X]   = q_pos * dt * dt;   // dt² scaling: pos uncertainty from integrating vel noise
  Q[IDX_Y][IDX_Y]   = q_pos * dt * dt;
  Q[IDX_Z][IDX_Z]   = q_pos * dt * dt;
  // Velocity noise: independent accel noise + configured vel process noise (add in quadrature)
  Q[IDX_VX][IDX_VX] = (q_vel + cfg.accel_noise_mps2 * cfg.accel_noise_mps2) * dt;
  Q[IDX_VY][IDX_VY] = (q_vel + cfg.accel_noise_mps2 * cfg.accel_noise_mps2) * dt;
  Q[IDX_VZ][IDX_VZ] = (q_vel + cfg.accel_noise_mps2 * cfg.accel_noise_mps2) * dt;
  // Yaw noise: gyro bias drift + configured heading process noise
  Q[IDX_YAW][IDX_YAW]           = (q_yaw + cfg.gyro_noise_rps * cfg.gyro_noise_rps) * dt;
  // Baro bias random walk: small diffusion per step keeps the bias state observable
  Q[IDX_BARO_BIAS][IDX_BARO_BIAS] = q_baro * dt;

  // Step 6: Covariance propagation — P = F·P·Fᵀ + Q
  float FP[EKF_N][EKF_N];
  float Ft[EKF_N][EKF_N];
  float FPFt[EKF_N][EKF_N];
  matMul8(F, ekf.P, FP);
  transpose8(F, Ft);
  matMul8(FP, Ft, FPFt);
  copyMat(FPFt, ekf.P);
  matAdd8(ekf.P, Q);  // P ← FPFᵀ + Q

  ekf.health.imu_ok = true;
}

// ---------------------------------------------------------------------
// gate1D — scalar normalized innovation squared (NIS) chi-squared test.
//
// For a single measurement dimension, the NIS is:
//   NIS = innovation² / innovationVar
// which follows a chi-squared distribution with 1 degree of freedom if the
// measurement model is correct. Rejecting when NIS > gateSigma² corresponds
// to a Gaussian σ test: NIS > 3.5² rejects at the ~99.98% confidence level.
//
// Returns false (reject) if innovationVar ≤ 0 (degenerate case) or NIS
// exceeds the gate threshold. Returns true (accept) otherwise.
```

```
// ---------------------------------------------------------------------------
static inline bool gate1D(float innovation, float innovationVar, float gateSigma) {
  if (innovationVar <= 0.0f) return false;
  float nis = (innovation * innovation) / innovationVar;
  return nis <= (gateSigma * gateSigma);
}

// ---------------------------------------------------------------------------
// updateScalar — single-axis Kalman update for a direct state measurement.
//
// Used for UWB X, Y, Z updates where H = eᵢ (unit vector selecting one state).
// For a scalar measurement z of state x[hIndex] with noise variance R:
//
//   innovation = z - x[hIndex]          (prediction error)
//   S = P[hIndex][hIndex] + R           (innovation covariance — scalar)
//   K[i] = P[i][hIndex] / S             (Kalman gain — full column / S)
//   x[i] += K[i] * innovation           (state update — all 8 states corrected)
//   P[i][j] -= K[i] * P[hIndex][j]      (covariance update — simplified Joseph form)
//
// The gain is the full column of P divided by the scalar S. This is the
// efficient form when H is a unit vector — no matrix-vector multiply needed.
// Yaw is re-wrapped after the update to stay in [-π, +π].
// ---------------------------------------------------------------------------
static inline void updateScalar(EkfState& ekf, int hIndex, float z, float R, float gateSigma, bool wrapYaw =
false) {
  float innovation = z - ekf.x[hIndex];
  if (wrapYaw) innovation = wrapPi(innovation);  // Handles yaw discontinuity at ±π

  // Innovation covariance S = H·P·Hᵀ + R; for unit-vector H this is just P[h][h] + R
  float S = ekf.P[hIndex][hIndex] + R;
  // Re-gate inside updateScalar to guard against repeated calls bypassing the outer gate
  if (!gate1D(innovation, S, gateSigma)) return;

  // Kalman gain: K = P·Hᵀ / S = column hIndex of P divided by scalar S
  float K[EKF_N];
  for (int i = 0; i < EKF_N; i++) K[i] = ekf.P[i][hIndex] / S;
```

```
  // State correction: x += K * innovation — propagates through all 8 states via cross-covariance
  for (int i = 0; i < EKF_N; i++) ekf.x[i] += K[i] * innovation;
  ekf.x[IDX_YAW] = wrapPi(ekf.x[IDX_YAW]);  // Always re-wrap yaw after any state update

  // Covariance update: P = (I - K·H)·P simplified for unit-vector H.
  // P_new[i][j] = P[i][j] - K[i] * P[hIndex][j]
  // Uses a local copy to avoid using partially updated rows in later iterations.
  float Pnew[EKF_N][EKF_N];
  copyMat(ekf.P, Pnew);
  for (int i = 0; i < EKF_N; i++) {
    for (int j = 0; j < EKF_N; j++) {
      Pnew[i][j] -= K[i] * ekf.P[hIndex][j];
    }
  }
  copyMat(Pnew, ekf.P);
}

// ---------------------------------------------------------------------
// updateFromUwb — fuse a UWB ranging measurement into the EKF.
//
// X, Y, Z are updated independently as three separate scalar updates.
// Each axis has its own innovation gate so a bad Z reading (e.g., multipath
// on a ground-reflected path) does not corrupt the X/Y update.
// has_x/y/z flags allow 2D-only anchor configurations to skip axes.
// Measurement noise R = std² for each axis (horizontal and vertical can differ).
//
// Returns true if at least one axis was accepted.
// ---------------------------------------------------------------------
inline bool updateFromUwb(EkfState& ekf, const EkfConfig& cfg, const UwbSample& uwb, Stream* dbg =
nullptr) {
  bool accepted = false;

  if (uwb.has_x) {
    float R = uwb.std_xy_m * uwb.std_xy_m;
    float innovation = uwb.x_m - ekf.x[IDX_X];
    float S = ekf.P[IDX_X][IDX_X] + R;
    bool ok = gate1D(innovation, S, cfg.uwb_gate_sigma);
```

```
  if (dbg) dbg->printf("EKF UWB X innov=%.3f S=%.3f ok=%d\n", innovation, S, ok ? 1 : 0);
  if (ok) {
    updateScalar(ekf, IDX_X, uwb.x_m, R, cfg.uwb_gate_sigma, false);
    accepted = true;
  }
 }

 if (uwb.has_y) {
  float R = uwb.std_xy_m * uwb.std_xy_m;
  float innovation = uwb.y_m - ekf.x[IDX_Y];
  float S = ekf.P[IDX_Y][IDX_Y] + R;
  bool ok = gate1D(innovation, S, cfg.uwb_gate_sigma);
  if (dbg) dbg->printf("EKF UWB Y innov=%.3f S=%.3f ok=%d\n", innovation, S, ok ? 1 : 0);
  if (ok) {
    updateScalar(ekf, IDX_Y, uwb.y_m, R, cfg.uwb_gate_sigma, false);
    accepted = true;
  }
 }

 if (uwb.has_z) {
  float R = uwb.std_z_m * uwb.std_z_m;
  float innovation = uwb.z_m - ekf.x[IDX_Z];
  float S = ekf.P[IDX_Z][IDX_Z] + R;
  bool ok = gate1D(innovation, S, cfg.uwb_gate_sigma);
  if (dbg) dbg->printf("EKF UWB Z innov=%.3f S=%.3f ok=%d\n", innovation, S, ok ? 1 : 0);
  if (ok) {
    updateScalar(ekf, IDX_Z, uwb.z_m, R, cfg.uwb_gate_sigma, false);
    accepted = true;
  }
 }

 ekf.last_uwb_us = uwb.t_us;
 ekf.health.uwb_ok = accepted;
 ekf.health.uwb_timeout = false;
 return accepted;
}
```

```
// ---------------------------------------------------------------------------
// updateFromBaro — fuse a barometer altitude measurement into the EKF.
//
// The barometer measures (Z + BARO_BIAS), not Z directly. The measurement
// model is H = [0,0,1,0,0,0,0,1] — a two-element row vector selecting both
// IDX_Z and IDX_BARO_BIAS. This makes it a non-scalar update that cannot
// use updateScalar(), so the gain and covariance update are computed manually.
//
// Innovation covariance S = H·P·Hᵀ + R, which expands to:
//   S = P[Z][Z] + P[BIAS][BIAS] + 2·P[Z][BIAS] + baro.std²
// The cross term 2·P[Z][BIAS] is important: if Z and BARO_BIAS are
// positively correlated, S is larger (less certain), reducing the gain.
//
// Kalman gain: K[i] = (P[i][Z] + P[i][BIAS]) / S
// This correctly weights both the altitude and bias columns of P.
//
// After the state update, BARO_BIAS is driven to absorb long-term pressure
// offset drift (e.g., temperature changes, weather) while Z tracks true
// altitude. The health flag baro_bias_large is set if |BARO_BIAS| > threshold.
// ---------------------------------------------------------------------------
inline bool updateFromBaro(EkfState& ekf, const EkfConfig& cfg, const BaroSample& baro, Stream* dbg =
nullptr) {
  // Predicted baro reading = estimated altitude + estimated bias
  float predicted  = ekf.x[IDX_Z] + ekf.x[IDX_BARO_BIAS];
  float innovation = baro.altitude_m - predicted;

  // Innovation covariance for H = [... 1 ... 1] at IDX_Z and IDX_BARO_BIAS
  float S = ekf.P[IDX_Z][IDX_Z] +
            ekf.P[IDX_BARO_BIAS][IDX_BARO_BIAS] +
            2.0f * ekf.P[IDX_Z][IDX_BARO_BIAS] +  // cross-covariance term
            baro.std_m * baro.std_m;              // measurement noise R

  bool ok = gate1D(innovation, S, cfg.baro_gate_sigma);
  if (dbg) dbg->printf("EKF BARO innov=%.3f S=%.3f ok=%d\n", innovation, S, ok ? 1 : 0);
  if (!ok) {
    ekf.last_baro_us = baro.t_us;  // Still update timestamp so timeout doesn't trigger
    return false;
```

```
  }

  // Kalman gain: K = P·Hᵀ / S, where Hᵀ = [col_Z + col_BIAS]
  float K[EKF_N];
  for (int i = 0; i < EKF_N; i++) {
    K[i] = (ekf.P[i][IDX_Z] + ekf.P[i][IDX_BARO_BIAS]) / S;
  }

  // State update: x += K * innovation — all 8 states adjusted via cross-covariance
  for (int i = 0; i < EKF_N; i++) ekf.x[i] += K[i] * innovation;
  ekf.x[IDX_YAW] = wrapPi(ekf.x[IDX_YAW]);

  // Covariance update: P_new[i][j] = P[i][j] - K[i] * (H·P)[j]
  // (H·P)[j] = P[IDX_Z][j] + P[IDX_BARO_BIAS][j] — the effective measurement row of P
  float Pnew[EKF_N][EKF_N];
  copyMat(ekf.P, Pnew);
  for (int i = 0; i < EKF_N; i++) {
    for (int j = 0; j < EKF_N; j++) {
      float HPj = ekf.P[IDX_Z][j] + ekf.P[IDX_BARO_BIAS][j];
      Pnew[i][j] -= K[i] * HPj;
    }
  }
  copyMat(Pnew, ekf.P);

  ekf.last_baro_us = baro.t_us;
  ekf.health.baro_ok = true;
  // Flag if the estimated bias is suspiciously large — may indicate sensor fault
  ekf.health.baro_bias_large = fabsf(ekf.x[IDX_BARO_BIAS]) > cfg.baro_bias_large_threshold_m;
  return true;
}

// ---------------------------------------------------------------------
// updateTimeouts — call once per loop to refresh health timeout flags.
// If no UWB or baro sample has arrived within the configured window,
// the corresponding health flags are cleared so callers can degrade fusion
// weights or raise a warning to the user.
// ---------------------------------------------------------------------
```

```
inline void updateTimeouts(EkfState& ekf, const EkfConfig& cfg, uint32_t now_us) {
  if (ekf.last_uwb_us == 0 || (now_us - ekf.last_uwb_us) > cfg.uwb_timeout_us) {
    ekf.health.uwb_timeout = true;
    ekf.health.uwb_ok = false;
  }
  if (ekf.last_baro_us == 0 || (now_us - ekf.last_baro_us) > cfg.baro_timeout_us) {
    ekf.health.baro_ok = false;
  }
  // Re-check bias magnitude each cycle — bias can drift back within threshold after correction
  ekf.health.baro_bias_large = fabsf(ekf.x[IDX_BARO_BIAS]) > cfg.baro_bias_large_threshold_m;
}

// ---------------------------------------------------------------------------
// getPoseEstimate — extract the current state into a caller-friendly struct.
//
// pos_confidence and vel_confidence use the reciprocal form 1/(1 + variance),
// mapping [0, ∞) variance to (1, 0] confidence. This gives ~0.5 confidence
// at 1 m² position variance and approaches 0 as covariance diverges (e.g.,
// after extended dead reckoning without UWB corrections).
// ---------------------------------------------------------------------------
inline PoseEstimate getPoseEstimate(const EkfState& ekf) {
  PoseEstimate p{};
  p.x_m        = ekf.x[IDX_X];
  p.y_m        = ekf.x[IDX_Y];
  p.z_m        = ekf.x[IDX_Z];
  p.vx_mps     = ekf.x[IDX_VX];
  p.vy_mps     = ekf.x[IDX_VY];
  p.vz_mps     = ekf.x[IDX_VZ];
  p.roll_rad   = ekf.roll_rad;   // From complementary filter, not EKF state
  p.pitch_rad  = ekf.pitch_rad;
  p.yaw_rad    = ekf.x[IDX_YAW];
  p.baro_bias_m = ekf.x[IDX_BARO_BIAS];

  // Scalar confidence: mean of the three diagonal position variances, mapped to [0,1]
  float posVar = (ekf.P[IDX_X][IDX_X] + ekf.P[IDX_Y][IDX_Y] + ekf.P[IDX_Z][IDX_Z]) / 3.0f;
  float velVar = (ekf.P[IDX_VX][IDX_VX] + ekf.P[IDX_VY][IDX_VY] + ekf.P[IDX_VZ][IDX_VZ]) / 3.0f;
  p.pos_confidence = 1.0f / (1.0f + posVar);
```

```
  p.vel_confidence = 1.0f / (1.0f + velVar);
  return p;
}

// Compact one-line debug dump — prints pose, bias, diagonal P, and health flags.
static inline void printDebug(Stream& s, const EkfState& ekf) {
  PoseEstimate p = getPoseEstimate(ekf);
  s.printf(
    "EKF pos=(%.2f,%.2f,%.2f) vel=(%.2f,%.2f,%.2f) rpy=(%.1f,%.1f,%.1f) bias=%.2f Pxyz=(%.3f,%.3f,%.3f)
uwb_ok=%d baro_ok=%d uwb_to=%d\n",
    p.x_m, p.y_m, p.z_m,
    p.vx_mps, p.vy_mps, p.vz_mps,
    p.roll_rad  * 180.0f / PI,
    p.pitch_rad * 180.0f / PI,
    p.yaw_rad   * 180.0f / PI,
    p.baro_bias_m,
    ekf.P[IDX_X][IDX_X], ekf.P[IDX_Y][IDX_Y], ekf.P[IDX_Z][IDX_Z],
    ekf.health.uwb_ok      ? 1 : 0,
    ekf.health.baro_ok     ? 1 : 0,
    ekf.health.uwb_timeout ? 1 : 0
  );
}

/*
Example usage:

  #include "handheld_ekf.h"
  using namespace HandheldEkf;

  EkfState ekf;
  EkfConfig cfg;

  void setup() {
    Serial.begin(115200);
    cfg = defaultConfig();
    initEkf(ekf, cfg, 0.0f, 0.0f, 0.0f, 0.0f);
  }
```

```
 void loop() {
   ImuSample imu;
   if (readImu(imu)) predictFromImu(ekf, cfg, imu);

   UwbSample uwb;
   if (readUwb(uwb)) updateFromUwb(ekf, cfg, uwb, &Serial);

   BaroSample baro;
   if (readBaro(baro)) updateFromBaro(ekf, cfg, baro, &Serial);

   updateTimeouts(ekf, cfg, micros());

   static uint32_t lastPrint = 0;
   if (millis() - lastPrint > 200) {
     lastPrint = millis();
     printDebug(Serial, ekf);
   }
 }
*/

} // namespace HandheldEkf

#endif
```

5. Scan Mode Header

```
#ifndef SCAN_MODE_H
#define SCAN_MODE_H

#include <Arduino.h>
#include <math.h>
#include <string.h>

// ---------------------------------------------------------------------------
// scan_mode.h — Multi-phase sensor fusion scan engine
//
// Architecture overview:
```

```
//   SCAN_INIT → SCAN_RADAR → SCAN_IR → SCAN_POSITION → SCAN_FUSION → SCAN_DONE
//
// Phase purposes:
//   RADAR    : Collect vitals (HR/BR) and presence from the mmWave radar.
//              This radar is noisy so the scoring is intentionally lenient —
//              any HR or BR reading in a live human range counts as positive
//              signal. When BLE vitals from an external Waveshare device are
//              available they are injected here via the same input fields.
//   IR       : Classify the thermal image from the MLX90640. This is the most
//              reliable sensor on the platform for human detection.
//   POSITION : Collect location from UWB ranging, GNSS, and/or INS dead
//              reckoning. Used to tag detections with a position estimate.
//   FUSION   : Weight-blend radar, IR, and position confidences into one
//              FusionDecision. IR carries the most weight since it's the most
//              reliable. Radar is a corroborating signal.
//
// Usage: call beginScan() once, then call updateScanMode() every loop iteration
// while active. Results accumulate in ScanRuntimeState.
// ---------------------------------------------------------------------------

namespace ScanMode {

// ---------------------------------------------------------------------------
// ScanState — current phase of the scan state machine.
// States advance linearly; ABORT can be entered from any active state.
// ---------------------------------------------------------------------------
enum ScanState : uint8_t {
  SCAN_IDLE = 0,    // not running — waiting for beginScan()
  SCAN_INIT,        // brief settle period before radar phase begins
  SCAN_RADAR,       // collecting mmWave vitals and presence
  SCAN_IR,          // capturing and classifying thermal frame
  SCAN_POSITION,    // acquiring UWB/GNSS/INS position fix
  SCAN_FUSION,      // blending results into final decision
  SCAN_DONE,        // scan complete — results are latched and stable
  SCAN_ABORT        // scan was cancelled mid-run
};
```

```
// ---------------------------------------------------------------------------
// FusionDecision — final verdict produced by fuseResults().
// Ordered by ascending confidence so comparisons work with >= operators.
// ---------------------------------------------------------------------------
enum FusionDecision : uint8_t {
  FUSION_NO_HUMAN = 0,    // no significant evidence of a human
  FUSION_POSSIBLE_HUMAN,  // weak signal — investigate further
  FUSION_LIKELY_HUMAN,    // moderate confidence — IR or radar positive
  FUSION_HUMAN_DETECTED   // high confidence — multiple sensors agree
};

// ---------------------------------------------------------------------------
// RadarScanResult — accumulated output from the SCAN_RADAR phase.
// All fields are updated each frame via latchRadarResult().
// ---------------------------------------------------------------------------
struct RadarScanResult {
  bool completed;              // true when phase exited normally or timed out
  bool timedOut;               // true if radarMaxDurationMs was reached
  bool compromisedBySafety;    // USS < RADAR_STOP_THRESH_CM during this phase
  bool pausedBySafety;         // same condition — kept separate for display use
  bool motionDetected;         // TLV motion flag from latest radar frame
  bool presenceDetected;       // TLV presence flag from latest radar frame
  bool breathingValid;         // breathing BPM passed range and artifact filter
  bool heartValid;             // heart rate BPM passed range and artifact filter
  float breathingBpm;          // current filtered breathing rate in BPM
  float heartRateBpm;          // current filtered heart rate in BPM
  float motionPresenceScore;   // 0-1 radar motion/presence signal strength
  float radarConfidence;       // 0-1 overall radar confidence from computeRadarConfidence()
  float signalQuality;         // 0-1 radar return SNR/quality
  float motionStability;       // 0-1 consistency of motion signature over time
  float breathingVariance;     // 0-1 frame-to-frame BR variability (lower = better)
  float heartVariance;         // 0-1 frame-to-frame HR variability (lower = better)
  float breathingConsistency;  // 0-1 how regularly the radar sees BR (higher = better)
  float heartConsistency;      // 0-1 how regularly the radar sees HR (higher = better)
  uint16_t validBreathingSamples; // count of frames where breathingValid was true
  uint16_t validHeartSamples;     // count of frames where heartValid was true
  uint16_t presenceFrames;        // count of frames where presenceDetected was true
```

```
  uint32_t phaseStartMs;       // millis() when SCAN_RADAR was entered
  uint32_t phaseElapsedMs;     // millis elapsed since phaseStartMs
  char warning[64];            // human-readable warning if safety event occurred
  char debug[160];             // formatted debug string of key signal values
};

// ---------------------------------------------------------------------------
// IrScanResult — output from the SCAN_IR phase.
// Populated once via latchIrResult() when the IR phase exits.
// ---------------------------------------------------------------------------
struct IrScanResult {
  bool completed;              // true when phase exited normally or timed out
  bool timedOut;               // true if irMaxDurationMs was reached
  bool humanLike;              // thermalHumanDetected flag from the detector
  float humanPresenceScore;    // 0-1 thermal detector human presence score
  float confidenceScore;       // 0-1 raw thermal detector confidence (matches THERMAL screen C)
  float categoryScores[10];    // per-class scores from the thermal classifier
  int bboxX, bboxY;            // top-left corner of the detected blob bounding box
  int bboxW, bboxH;            // width and height of the blob bounding box in pixels
  float centroidX, centroidY;  // centroid of the thermal blob in sensor pixel space
  float aspectRatio;           // blob width/height — useful for body shape detection
  int hotPixelCount;           // number of pixels above the human-temperature threshold
  float meanTargetTemp;        // mean temperature of hot pixels in °C
  float maxTargetTemp;         // peak temperature of any hot pixel in °C
  float contrastAboveBackground;// how much hotter the blob is than local background
  uint32_t phaseStartMs;       // millis() when SCAN_IR was entered
  uint32_t phaseElapsedMs;     // millis elapsed since phaseStartMs
  char topCategory[32];        // name of the highest-scoring thermal classifier class
  char debug[192];             // raw debug string from the thermal detector
};

// ---------------------------------------------------------------------------
// PositionScanResult — output from the SCAN_POSITION phase.
// Records the best available position fix at time of detection.
// ---------------------------------------------------------------------------
struct PositionScanResult {
  bool completed;              // true when phase exited normally or timed out
```

```
  bool timedOut;              // true if positionMaxDurationMs was reached
  bool uwbValid;              // UWB ranging produced a valid distance reading
  bool gnssValid;             // GPS/GNSS had a valid fix
  bool insValid;              // IMU dead-reckoning position is available
  float uwbRangeM;            // UWB distance to anchor in meters (NAN if invalid)
  float relX_m;               // relative east position from origin in meters
  float relY_m;               // relative north position from origin in meters
  float relZ_m;               // relative altitude from baseline in meters
  float rollDeg;              // roll at time of position capture in degrees
  float pitchDeg;             // pitch at time of position capture in degrees
  float yawDeg;               // yaw/heading at time of position capture in degrees
  float positionConfidence;   // 0-1 composite position confidence
  double lat;                 // latitude if GNSS valid (degrees)
  double lon;                 // longitude if GNSS valid (degrees)
  uint32_t phaseStartMs;      // millis() when SCAN_POSITION was entered
  uint32_t phaseElapsedMs;    // millis elapsed since phaseStartMs
  char debug[160];            // formatted debug string of position state
};

// ---------------------------------------------------------------------------
// FusionScanResult — final output of fuseResults().
// Blends radar, IR, and position into a single decision with weights chosen
// based on which confidence tier each sensor reached.
// ---------------------------------------------------------------------------
struct FusionScanResult {
  bool completed;             // true after fuseResults() has run
  FusionDecision decision;    // final verdict enum value
  float radarWeight;          // weight applied to radar confidence (0-1)
  float irWeight;             // weight applied to IR confidence (0-1)
  float positionWeight;       // weight applied to position confidence (0-1)
  float radarContribution;    // radar * radarWeight component of overall score
  float irContribution;       // ir * irWeight component of overall score
  float positionContribution; // pos * positionWeight component of overall score
  float overallConfidence;    // sum of contributions, clamped to 0-1
  bool showPosition;          // true if any position source was valid
  char decisionText[32];      // human-readable decision string for display
  char warning[96];           // populated if radar was compromised by USS
```

```
  char debug[192];              // per-weight debug breakdown string
};

// ---------------------------------------------------------------------------
// ScanConfig — tunable parameters for the scan engine.
// Call defaultConfig() to get a pre-tuned starting point, then adjust fields
// before passing to updateScanMode().
// ---------------------------------------------------------------------------
struct ScanConfig {
  // Phase time budgets
  uint32_t initTimeoutMs;          // how long to wait in SCAN_INIT before starting radar
  uint32_t radarMaxDurationMs;     // maximum time allowed for the radar phase
  uint32_t irMaxDurationMs;        // maximum time allowed for the IR phase
  uint32_t positionMaxDurationMs;  // maximum time allowed for the position phase
  uint32_t fusionHoldMs;           // how long to show the fusion result before SCAN_DONE
  uint32_t summaryRefreshMs;       // minimum interval between render callbacks

  // Radar thresholds — intentionally low because this sensor is noisy.
  // Any HR or BR in a physiological range is treated as positive signal.
  float radarDetectedThreshold;    // radar confidence to call FUSION_HUMAN_DETECTED
  float radarLikelyThreshold;      // radar confidence for FUSION_LIKELY_HUMAN
  float radarPossibleThreshold;    // radar confidence for FUSION_POSSIBLE_HUMAN

  // IR thresholds — higher than radar because thermal is more reliable
  float irDetectedThreshold;
  float irLikelyThreshold;
  float irPossibleThreshold;

  // Fusion thresholds — override signal-based decision with overall confidence
  float fusionDetectedThreshold;
  float fusionLikelyThreshold;
  float fusionPossibleThreshold;

  // Fusion weights — three sets chosen based on which tier each sensor reached.
  // IR carries more weight than radar because it is the more reliable sensor here.
  // BLE-boosted radar vitals elevate the radar weight automatically via the
  // radarBreathingValid/radarHeartValid flags in ScanSensorInputs.
```

```
  float radarWeightDetected;       // radar weight when both sensors are at "detected" tier
  float irWeightDetected;
  float positionWeightDetected;
  float radarWeightLikely;         // weights when IR is at "likely" tier
  float irWeightLikely;
  float positionWeightLikely;
  float radarWeightPossible;       // weights when neither sensor is confident
  float irWeightPossible;
  float positionWeightPossible;

  // Vital sign quality gates — used as soft penalty thresholds, not hard rejections
  float breathingJumpLimitBpm;     // max single-frame BR change before flagging
  float heartJumpLimitBpm;         // max single-frame HR change before flagging
  float minMotionStability;        // below this, stability penalty is applied
  float minSignalQuality;          // below half this, signal quality penalty applied
  float minBreathingConsistency;   // below this, BR consistency penalty applied
  float minHeartConsistency;       // below this, HR consistency penalty applied

  bool collectPositionOnNegative;  // if true, always collect position even on negative result
};

// ---------------------------------------------------------------------------
// ScanRuntimeState — live state of a running scan.
// Passed by reference into updateScanMode() every loop iteration.
// Do not read results until state == SCAN_DONE.
// ---------------------------------------------------------------------------
struct ScanRuntimeState {
  ScanState state;             // current phase
  ScanState previousState;     // previous phase (for transition logic)
  bool active;                 // true between beginScan() and SCAN_DONE/ABORT
  bool cancelRequested;        // set by requestCancel() — triggers SCAN_ABORT
  bool phaseEntered;           // true after enterPhase() has run for current state
  bool renderDirty;            // true when render callbacks should fire immediately
  bool summaryLatched;         // true when SCAN_DONE or SCAN_ABORT has been reached
  uint32_t scanStartMs;        // millis() when beginScan() was called
  uint32_t stateStartMs;       // millis() when current state was entered
  uint32_t lastUpdateMs;       // millis() of last updateScanMode() call
```

```
  uint32_t lastRenderMs;        // millis() of last render callback fire
  uint32_t lastDebugMs;         // millis() of last serialDebug callback fire
  uint16_t progressPermille;    // 0-1000 progress within current phase for progress bar
  RadarScanResult radar;        // accumulated radar phase results
  IrScanResult ir;              // accumulated IR phase results
  PositionScanResult position;  // accumulated position phase results
  FusionScanResult fusion;      // final fusion output
};

// ---------------------------------------------------------------------------
// ScanSensorInputs — all live sensor readings passed to updateScanMode().
// Fill this struct each loop iteration before calling updateScanMode().
//
// BLE integration note: when Waveshare vitals arrive over BLE, inject them
// directly into radarBreathingBpm/radarHeartRateBpm and set the Valid flags
// true. The radar confidence scorer treats BLE-sourced vitals identically to
// locally-measured ones, so BLE data automatically boosts the radar score
// without any separate code path.
// ---------------------------------------------------------------------------
struct ScanSensorInputs {
  // USS proximity guard — if true, radar FOV may be obstructed
  bool ussTooClose;

  // mmWave radar vitals and presence
  // These fields accept either locally-measured radar values or BLE-injected
  // vitals from an external sensor — the scorer does not distinguish the source.
  bool radarMotionDetected;         // TLV motion flag from radar frame
  bool radarPresenceDetected;       // TLV presence flag from radar frame
  bool radarBreathingValid;         // true if BR is in a plausible range
  bool radarHeartValid;             // true if HR is in a plausible range
  float radarBreathingBpm;          // breathing rate in BPM (local or BLE-injected)
  float radarHeartRateBpm;          // heart rate in BPM (local or BLE-injected)
  float radarMotionPresenceScore;   // 0-1 radar presence signal strength
  float radarSignalQuality;         // 0-1 radar return SNR
  float radarMotionStability;       // 0-1 motion consistency over time
  float radarBreathingVariance;     // 0-1 BR frame-to-frame variability
  float radarHeartVariance;         // 0-1 HR frame-to-frame variability
```

```
  float radarBreathingConsistency;  // 0-1 how often radar produces valid BR
  float radarHeartConsistency;      // 0-1 how often radar produces valid HR

  // MLX90640 thermal camera outputs
  bool irHumanLike;                 // thermalHumanDetected from the detector
  float irHumanPresenceScore;       // 0-1 thermal human presence score
  float irConfidenceScore;          // 0-1 thermal classifier confidence
  const float* irCategoryScores;    // pointer to per-class score array (10 classes)
  int irBBoxX, irBBoxY;             // blob bounding box origin in pixel space
  int irBBoxW, irBBoxH;             // blob bounding box dimensions in pixels
  float irCentroidX, irCentroidY;   // blob centroid in sensor pixel space
  float irAspectRatio;              // blob width/height ratio
  int irHotPixelCount;              // count of pixels above temperature threshold
  float irMeanTargetTemp;           // mean blob temperature in °C
  float irMaxTargetTemp;            // peak blob temperature in °C
  float irContrastAboveBackground;  // blob temp minus local background temp in °C
  const char* irTopCategory;        // name of highest-scoring class
  const char* irDebug;              // raw debug string from thermal detector

  // Position inputs — from UWB, GNSS, and/or INS dead reckoning
  bool uwbValid;                    // UWB ranging produced a valid reading
  float uwbRangeM;                  // UWB distance to anchor in meters
  bool gnssValid;                   // GNSS has a valid fix
  double lat;                       // latitude in degrees (valid only if gnssValid)
  double lon;                       // longitude in degrees (valid only if gnssValid)
  bool insValid;                    // IMU dead-reckoning is available
  float relX_m;                     // relative east displacement from origin in meters
  float relY_m;                     // relative north displacement from origin in meters
  float relZ_m;                     // relative altitude from baseline in meters
  float rollDeg;                    // current roll in degrees
  float pitchDeg;                   // current pitch in degrees
  float yawDeg;                     // current yaw/heading in degrees
  float positionConfidence;         // 0-1 overall position source confidence
};

// ---------------------------------------------------------------------------
// ScanCallbacks — optional function pointers for render and phase lifecycle.
```

```
// All fields default to nullptr — only populate the ones you need.
// Render callbacks fire at summaryRefreshMs intervals or when renderDirty is set.
// ---------------------------------------------------------------------------
struct ScanCallbacks {
  void (*enterRadarPhase)();       // called once when SCAN_RADAR is entered
  void (*updateRadarPhase)();      // called every updateScanMode() in SCAN_RADAR
  void (*exitRadarPhase)();        // called once when SCAN_RADAR is exited
  void (*enterIrPhase)();
  void (*updateIrPhase)();
  void (*exitIrPhase)();
  void (*enterPositionPhase)();
  void (*updatePositionPhase)();
  void (*exitPositionPhase)();
  void (*renderRadarPhase)(const ScanRuntimeState&, uint16_t progressPermille);
  void (*renderIrPhase)(const ScanRuntimeState&, uint16_t progressPermille);
  void (*renderPositionPhase)(const ScanRuntimeState&, uint16_t progressPermille);
  void (*renderFusionPhase)(const ScanRuntimeState&, uint16_t progressPermille);
  void (*renderDonePhase)(const ScanRuntimeState&);   // called when SCAN_DONE is reached
  void (*renderAbortPhase)(const ScanRuntimeState&);  // called when SCAN_ABORT is reached
  void (*renderProgressBar)(ScanState state, uint16_t progressPermille);
  void (*serialDebug)(const ScanRuntimeState&);       // called every 1000ms for serial logging
};

// ---------------------------------------------------------------------------
// Utility math helpers
// ---------------------------------------------------------------------------

// clamp01 — clamps a float to [0.0, 1.0]
static inline float clamp01(float v) {
  if (v < 0.0f) return 0.0f;
  if (v > 1.0f) return 1.0f;
  return v;
}

// safeNorm — linearly maps [lo, hi] → [0, 1], clamped
static inline float safeNorm(float value, float lo, float hi) {
  if (hi <= lo) return value >= hi ? 1.0f : 0.0f;
```

```
  return clamp01((value - lo) / (hi - lo));
}

// ---------------------------------------------------------------------------
// String helpers
// ---------------------------------------------------------------------------

static inline const char* stateName(ScanState s) {
  switch (s) {
    case SCAN_IDLE:     return "IDLE";
    case SCAN_INIT:     return "INIT";
    case SCAN_RADAR:    return "RADAR";
    case SCAN_IR:       return "IR";
    case SCAN_POSITION: return "POSITION";
    case SCAN_FUSION:   return "FUSION";
    case SCAN_DONE:     return "DONE";
    case SCAN_ABORT:    return "ABORT";
    default:            return "?";
  }
}

static inline const char* decisionName(FusionDecision d) {
  switch (d) {
    case FUSION_HUMAN_DETECTED:  return "Human detected";
    case FUSION_LIKELY_HUMAN:    return "Likely human";
    case FUSION_POSSIBLE_HUMAN:  return "Possible human";
    default:                     return "No human detected";
  }
}

// ---------------------------------------------------------------------------
// defaultConfig — returns a pre-tuned ScanConfig.
//
// Radar thresholds are intentionally low because this mmWave sensor is noisy
// and used as a corroborating sensor rather than the primary detector. Any HR
// or BR reading in a physiological range is treated as meaningful evidence.
//
```

```
// IR thresholds are higher because the MLX90640 thermal camera is the most
// reliable sensor on this platform for human detection.
//
// Fusion weights are IR-heavy for the same reason. Position carries a small
// bonus weight since a confirmed range/fix corroborates the detection.
// ---------------------------------------------------------------------------
static inline ScanConfig defaultConfig() {
  ScanConfig c{};
  c.initTimeoutMs          = 2000;
  // Radar runs Phase 1 (30 s) + Phase 2 (30 s) back-to-back driven by
  // RADAR_PHASE_MS in the main .ino, so the SCAN_RADAR state budget is 60 s
  // total. The previous 120 s was a legacy ceiling from when phases were
  // longer and made the on-screen phase timer count down against the wrong
  // total.
  c.radarMaxDurationMs     = 60000;
  c.irMaxDurationMs        = 6000;
  c.positionMaxDurationMs  = 5000;
  c.fusionHoldMs           = 1200;
  c.summaryRefreshMs       = 150;

  // Radar — loose thresholds to match sensor accuracy limitations
  c.radarDetectedThreshold  = 0.55f;
  c.radarLikelyThreshold    = 0.38f;
  c.radarPossibleThreshold  = 0.22f;

  // IR — tighter thresholds, thermal is more reliable
  c.irDetectedThreshold     = 0.75f;
  c.irLikelyThreshold       = 0.55f;
  c.irPossibleThreshold     = 0.35f;

  // Fusion — overall weighted score thresholds
  c.fusionDetectedThreshold = 0.78f;
  c.fusionLikelyThreshold   = 0.55f;
  c.fusionPossibleThreshold = 0.32f;

  // Fusion weights — IR-heavy since thermal is the primary detector
  c.radarWeightDetected     = 0.35f;
```

```
  c.irWeightDetected        = 0.55f;
  c.positionWeightDetected  = 0.10f;

  c.radarWeightLikely       = 0.25f;
  c.irWeightLikely          = 0.65f;
  c.positionWeightLikely    = 0.10f;

  // When neither sensor is confident, radar leads slightly since
  // presence/motion is cheaper to detect than a thermal blob
  c.radarWeightPossible     = 0.45f;
  c.irWeightPossible        = 0.45f;
  c.positionWeightPossible  = 0.10f;

  // Quality penalty thresholds — used as soft gates, not hard rejections
  c.breathingJumpLimitBpm   = 4.0f;
  c.heartJumpLimitBpm       = 12.0f;
  c.minMotionStability      = 0.45f;
  c.minSignalQuality        = 0.35f;
  c.minBreathingConsistency = 0.45f;
  c.minHeartConsistency     = 0.40f;

  c.collectPositionOnNegative = false;
  return c;
}

// ---------------------------------------------------------------------------
// Reset helpers — zero-initialize each result struct and set NAN sentinels
// ---------------------------------------------------------------------------

static inline void resetRadarResult(RadarScanResult& r) {
  memset(&r, 0, sizeof(r));
  r.breathingBpm  = NAN;
  r.heartRateBpm  = NAN;
}

static inline void resetIrResult(IrScanResult& r) {
  memset(&r, 0, sizeof(r));
```

```
}

static inline void resetPositionResult(PositionScanResult& r) {
  memset(&r, 0, sizeof(r));
  r.uwbRangeM = NAN;
  r.relX_m    = NAN;
  r.relY_m    = NAN;
  r.relZ_m    = NAN;
}

static inline void resetFusionResult(FusionScanResult& r) {
  memset(&r, 0, sizeof(r));
  r.decision = FUSION_NO_HUMAN;
  strncpy(r.decisionText, decisionName(r.decision), sizeof(r.decisionText) - 1);
}

static inline void resetRuntime(ScanRuntimeState& rt) {
  memset(&rt, 0, sizeof(rt));
  rt.state         = SCAN_IDLE;
  rt.previousState = SCAN_IDLE;
  resetRadarResult(rt.radar);
  resetIrResult(rt.ir);
  resetPositionResult(rt.position);
  resetFusionResult(rt.fusion);
}

// ---------------------------------------------------------------------------
// computeRadarConfidence — scores 0.0-1.0 how confident we are that
// the radar is detecting a live human.
//
// Design philosophy: this sensor (Waveshare HMMD mmWave) is noisy and
// frequently produces artifacts, especially at the known 50.2 bpm firmware
// ghost reading. The scorer is intentionally lenient — if the radar is picking
// up ANY heart rate or breathing rate in a physiological range, that is treated
// as meaningful positive evidence rather than requiring high consistency scores.
//
// Score breakdown (approximately):
```

```
//   Presence/motion signal  : ~30%  — graduated, not binary
//   Breathing rate in range : ~30%  — full score if 8-28 BPM, partial if 5-35
//   Heart rate in range     : ~30%  — full score if 50-120 BPM, partial 40-140
//   Both vitals valid bonus :  10%  — rewarded when BR and HR are both present
//
// Penalties (multiplicative, applied after score is built):
//   USS too close           : ×0.30 — radar FOV physically blocked
//   Very weak signal        : ×0.75 — if signal < half minSignalQuality threshold
//   No vitals and no presence: ×0.30 — effectively nothing to go on
//
// BLE integration: when Waveshare vitals are available over BLE, inject them
// into radarBreathingBpm/radarHeartRateBpm and set the Valid flags true before
// calling updateScanMode(). The scorer treats them identically to local readings.
// ----------------------------------------------------------------------------
static inline float computeRadarConfidence(const ScanSensorInputs& in, const ScanConfig& cfg) {
  float score = 0.0f;
  const bool radarHasEvidence =
    in.radarPresenceDetected ||
    in.radarMotionDetected ||
    in.radarBreathingValid ||
    in.radarHeartValid;

  // --- Presence and motion (~30%) ---
  // Graduated rather than binary — partial credit for weak but real signals
  score += clamp01(in.radarMotionPresenceScore) * 0.15f;
  score += clamp01(in.radarSignalQuality)       * 0.08f;
  score += clamp01(in.radarMotionStability)     * 0.02f;  // small — stability is unreliable on this sensor

  // Binary TLV flag bonuses — independent of the scored values above
  if (in.radarPresenceDetected) score += 0.07f;
  if (in.radarMotionDetected)   score += 0.07f;
  // Extra bonus when both presence and motion confirm simultaneously
  if (in.radarPresenceDetected && in.radarMotionDetected) score += 0.03f;

  // --- Breathing rate (~25%) ---
  // Any BR reading in a live human range is treated as positive evidence.
  // Consistency is a soft bonus — it matters but a single good reading still counts.
```

```
if (in.radarBreathingValid && isfinite(in.radarBreathingBpm)) {
  float bpmScore = 0.0f;
  float bpm = in.radarBreathingBpm;
  if      (bpm >= 8.0f  && bpm <= 28.0f) bpmScore = 1.0f;   // ideal physiological range
  else if (bpm >= 5.0f  && bpm <= 35.0f) bpmScore = 0.55f;  // marginal but plausible
  float consistencyBonus = clamp01(in.radarBreathingConsistency) * 0.30f;
  score += (bpmScore + consistencyBonus) * 0.25f;
  score += 0.05f;  // existence bonus — just having any valid BR reading is evidence
}

// --- Heart rate (~25%) ---
// Same philosophy as BR. Filter out the known 50.2 bpm firmware artifact
// that this mmWave module produces when no human is present.
if (in.radarHeartValid && isfinite(in.radarHeartRateBpm)) {
  float bpm = in.radarHeartRateBpm;
  float bpmScore = 0.0f;
  bool isArtifact = fabsf(bpm - 50.2f) < 1.5f;  // known ghost reading from firmware
  if (!isArtifact) {
    if      (bpm >= 50.0f && bpm <= 120.0f) bpmScore = 1.0f;   // normal resting to elevated
    else if (bpm >= 40.0f && bpm <= 140.0f) bpmScore = 0.55f;  // wide but physiologically possible
  }
  float consistencyBonus = clamp01(in.radarHeartConsistency) * 0.30f;
  score += (bpmScore + consistencyBonus) * 0.25f;
  // Bonus only if reading is real and in range — artifact readings get nothing
  if (!isArtifact && bpmScore > 0.0f) score += 0.05f;
}

// --- Both vitals present bonus (10%) ---
// When both BR and HR are valid simultaneously, that is much stronger evidence
// than either alone. This also naturally rewards BLE-injected vitals since
// the Waveshare provides both readings when it detects a human.
if (in.radarBreathingValid && in.radarHeartValid) {
  score += 0.10f;
}

// --- Penalties (multiplicative — applied last so they scale the full score) ---
```

```
  // USS too close: radar is physically blocked by the close object
  if (in.ussTooClose) score *= 0.30f;

  // Very weak signal: soft penalty, only applied below half the quality floor
  // to avoid penalizing moderately noisy but real signals
  if (in.radarSignalQuality < cfg.minSignalQuality * 0.5f) score *= 0.75f;

  // No evidence at all: presence, motion, and both vitals all absent
  // This catches cases where the radar is running but seeing nothing
  if (!in.radarBreathingValid && !in.radarHeartValid && !in.radarPresenceDetected) {
    score *= 0.30f;
  }

  // Project rule: if radar is actively returning any human-related data and the
  // module is not currently compromised by the close-range safety interlock,
  // treat that as a strong positive signal instead of allowing quality terms
  // to drag the score below the operator's expected floor.
  if (radarHasEvidence && !in.ussTooClose) {
    score = max(score, 0.80f);
  }

  return clamp01(score);
}

// ---------------------------------------------------------------------------
// computeIrConfidence — scores 0.0-1.0 how confident the thermal camera
// is that a human is present.
//
// Score breakdown:
//   humanPresenceScore : 55% — primary thermal detector output
//   confidenceScore    : 25% — secondary classifier confidence
//   topCategory score  : 10% — best-matching body-part class score
//   sizeFactor         : 10% — blob pixel count normalized to ~18 pixels
//
// IR is the most reliable sensor on this platform — higher confidence
// thresholds are set for it in defaultConfig() accordingly.
// ---------------------------------------------------------------------------
```

```
static inline float computeIrConfidence(const ScanSensorInputs& in) {
  const float rawConfidence = clamp01(in.irConfidenceScore);
  const float humanPresence = clamp01(in.irHumanPresenceScore);
  float topCategory = 0.0f;
  if (in.irCategoryScores) {
    for (int i = 0; i < 9; i++) {
      if (in.irCategoryScores[i] > topCategory) topCategory = in.irCategoryScores[i];
    }
  }
  // Normalize hot pixel count — 18 pixels is roughly a head-sized blob at 1-2m
  float sizeFactor = clamp01((float)in.irHotPixelCount / 18.0f);
  float score =
    humanPresence                * 0.55f +
    rawConfidence                * 0.25f +
    clamp01(topCategory)             * 0.10f +
    sizeFactor                   * 0.10f;

  // No artificial floor here: the scan thermal screen must mirror the raw
  // thermal-screen confidence so a stale peak can decay naturally instead of
  // being pinned to 100% by the blended heuristic.
  return clamp01(score);
}

// ---------------------------------------------------------------------------
// computePositionConfidence — scores 0.0-1.0 how reliable the position fix is.
// Used as a small tiebreaker weight in fusion, not as a primary detection signal.
//
// Score breakdown:
//   UWB valid  : 0.40 — centimeter-level ranging, most accurate indoors
//   GNSS valid : 0.35 — meter-level, only reliable outdoors
//   INS valid  : 0.25 — dead reckoning, drifts over time but always available
//
// The raw score is scaled by positionConfidence (0-1) from the caller, which
// reflects how long since the INS origin was set or GPS signal quality.
// ---------------------------------------------------------------------------
static inline float computePositionConfidence(const ScanSensorInputs& in) {
  float score = 0.0f;
```

```
  if (in.uwbValid)  score += 0.40f;
  if (in.gnssValid) score += 0.35f;
  if (in.insValid)  score += 0.25f;
  score *= clamp01(in.positionConfidence > 0.0f ? in.positionConfidence : 1.0f);
  return clamp01(score);
}

// ---------------------------------------------------------------------------
// latchRadarResult — snapshot the current sensor inputs into the radar result
// struct and recompute radarConfidence. Called every frame during SCAN_RADAR.
// ---------------------------------------------------------------------------
static inline void latchRadarResult(ScanRuntimeState& rt, const ScanSensorInputs& in, const ScanConfig&
cfg, uint32_t nowMs) {
  RadarScanResult& r = rt.radar;
  r.phaseElapsedMs       = nowMs - r.phaseStartMs;
  r.motionDetected       = in.radarMotionDetected;
  r.presenceDetected     = in.radarPresenceDetected;
  r.breathingValid       = in.radarBreathingValid;
  r.heartValid           = in.radarHeartValid;
  r.breathingBpm         = in.radarBreathingBpm;
  r.heartRateBpm         = in.radarHeartRateBpm;
  r.motionPresenceScore  = clamp01(in.radarMotionPresenceScore);
  r.signalQuality        = clamp01(in.radarSignalQuality);
  r.motionStability      = clamp01(in.radarMotionStability);
  r.breathingVariance    = clamp01(in.radarBreathingVariance);
  r.heartVariance        = clamp01(in.radarHeartVariance);
  r.breathingConsistency = clamp01(in.radarBreathingConsistency);
  r.heartConsistency     = clamp01(in.radarHeartConsistency);

  // Running counters — increment rather than overwrite so phase history is preserved
  if (in.radarBreathingValid) r.validBreathingSamples++;
  if (in.radarHeartValid)     r.validHeartSamples++;
  if (in.radarPresenceDetected) r.presenceFrames++;

  r.compromisedBySafety = in.ussTooClose;
  r.pausedBySafety      = in.ussTooClose;
```

```
  if (in.ussTooClose) {
    strncpy(r.warning, "USS too close, radar compromised", sizeof(r.warning) - 1);
  } else {
    r.warning[0] = '\0';
  }

  // Recompute confidence from the latest inputs — this runs every frame so the
  // displayed value always reflects the most recent sensor state
  r.radarConfidence = computeRadarConfidence(in, cfg);

  snprintf(r.debug, sizeof(r.debug),
    "prs=%.2f sig=%.2f mot=%.2f brC=%.2f hrC=%.2f brV=%.2f hrV=%.2f close=%d",
    r.motionPresenceScore, r.signalQuality, r.motionStability,
    r.breathingConsistency, r.heartConsistency,
    r.breathingVariance, r.heartVariance,
    in.ussTooClose ? 1 : 0);
}

// ---------------------------------------------------------------------------
// latchIrResult — snapshot thermal detector outputs into the IR result struct.
// Called once when the IR phase exits (either on detection or timeout).
// ---------------------------------------------------------------------------
static inline void latchIrResult(ScanRuntimeState& rt, const ScanSensorInputs& in, uint32_t nowMs) {
  IrScanResult& r    = rt.ir;
  r.phaseElapsedMs   = nowMs - r.phaseStartMs;
  r.humanLike        = in.irHumanLike;
  r.humanPresenceScore = clamp01(in.irHumanPresenceScore);
  r.confidenceScore  = clamp01(in.irConfidenceScore);  // exactly matches THERMAL screen C
  if (in.irCategoryScores) {
    for (int i = 0; i < 10; i++) r.categoryScores[i] = clamp01(in.irCategoryScores[i]);
  }
  r.bboxX            = in.irBBoxX;
  r.bboxY            = in.irBBoxY;
  r.bboxW            = in.irBBoxW;
  r.bboxH            = in.irBBoxH;
  r.centroidX        = in.irCentroidX;
  r.centroidY        = in.irCentroidY;
```

```
  r.aspectRatio      = in.irAspectRatio;
  r.hotPixelCount    = in.irHotPixelCount;
  r.meanTargetTemp   = in.irMeanTargetTemp;
  r.maxTargetTemp    = in.irMaxTargetTemp;
  r.contrastAboveBackground = in.irContrastAboveBackground;
  if (in.irTopCategory) strncpy(r.topCategory, in.irTopCategory, sizeof(r.topCategory) - 1);
  if (in.irDebug)       strncpy(r.debug, in.irDebug, sizeof(r.debug) - 1);
}

// ---------------------------------------------------------------------------
// latchPositionResult — snapshot position inputs into the position result struct.
// Called once when the position phase exits.
// ---------------------------------------------------------------------------
static inline void latchPositionResult(ScanRuntimeState& rt, const ScanSensorInputs& in, uint32_t nowMs) {
  PositionScanResult& r = rt.position;
  r.phaseElapsedMs   = nowMs - r.phaseStartMs;
  r.uwbValid         = in.uwbValid;
  r.gnssValid        = in.gnssValid;
  r.insValid         = in.insValid;
  r.uwbRangeM        = in.uwbRangeM;
  r.relX_m           = in.relX_m;
  r.relY_m           = in.relY_m;
  r.relZ_m           = in.relZ_m;
  r.rollDeg          = in.rollDeg;
  r.pitchDeg         = in.pitchDeg;
  r.yawDeg           = in.yawDeg;
  r.lat              = in.lat;
  r.lon              = in.lon;
  r.positionConfidence = computePositionConfidence(in);
  snprintf(r.debug, sizeof(r.debug),
    "uwb=%d gnss=%d ins=%d pc=%.2f x=%.2f y=%.2f yaw=%.1f",
    r.uwbValid ? 1 : 0, r.gnssValid ? 1 : 0, r.insValid ? 1 : 0,
    r.positionConfidence, r.relX_m, r.relY_m, r.yawDeg);
}

// ---------------------------------------------------------------------------
// fuseResults — blends radar, IR, and position into a FusionDecision.
```

```
//
// Weight selection is tier-based rather than fixed:
//   Both radar+IR at "detected" tier → use detected weights (IR-heavy: 35/55/10)
//   IR at "likely" tier              → use likely weights   (IR-heavy: 25/65/10)
//   Otherwise                        → use possible weights (balanced: 45/45/10)
//
// Signal-based decision is set first from individual sensor thresholds, then
// overridden by the overall weighted confidence if it crosses a fusion threshold.
// This means a single strong sensor can carry the decision even if the other is weak.
//
// Warning is propagated if radar was compromised by a USS proximity event,
// which tells the operator the radar data may be unreliable for this scan.
// ----------------------------------------------------------------------------
static inline void fuseResults(ScanRuntimeState& rt, const ScanConfig& cfg) {
  FusionScanResult& f = rt.fusion;
  const float radar = rt.radar.radarConfidence;
  const float ir    = rt.ir.confidenceScore;
  const float pos   = rt.position.positionConfidence;

  // Select weight set based on which confidence tier each sensor reached
  if (radar >= cfg.radarDetectedThreshold && ir >= cfg.irDetectedThreshold) {
    // Both sensors at "detected" tier — use detected weights
    f.radarWeight    = cfg.radarWeightDetected;
    f.irWeight       = cfg.irWeightDetected;
    f.positionWeight = cfg.positionWeightDetected;
  } else if (ir >= cfg.irLikelyThreshold) {
    // IR is strong but radar may be weak — shift more weight to IR
    f.radarWeight    = cfg.radarWeightLikely;
    f.irWeight       = cfg.irWeightLikely;
    f.positionWeight = cfg.positionWeightLikely;
  } else {
    // Neither sensor is confident — use possible weights
    f.radarWeight    = cfg.radarWeightPossible;
    f.irWeight       = cfg.irWeightPossible;
    f.positionWeight = cfg.positionWeightPossible;
  }
```

```
// Compute weighted contributions and overall confidence
f.radarContribution    = radar * f.radarWeight;
f.irContribution       = ir    * f.irWeight;
f.positionContribution = pos   * f.positionWeight;
// Position should never suppress a detection verdict. Keep it available for
// display, but base the human/no-human confidence on radar + thermal only.
const float detectionWeight = max(0.01f, f.radarWeight + f.irWeight);
f.overallConfidence = clamp01((f.radarContribution + f.irContribution) / detectionWeight);
f.showPosition         = rt.position.uwbValid || rt.position.gnssValid || rt.position.insValid;

// Decision is taken directly from the overall weighted score:
//   >= 0.70  -> Human detected
//   >= 0.40  -> Possible human
//    < 0.40  -> No human
// This matches the three on-screen labels and replaces the older multi-rule
// per-sensor tiering (which could flip between LIKELY/POSSIBLE even when the
// blended score was clearly above 70%).
if (f.overallConfidence >= 0.70f) {
  f.decision = FUSION_HUMAN_DETECTED;
} else if (f.overallConfidence >= 0.40f) {
  f.decision = FUSION_POSSIBLE_HUMAN;
} else {
  f.decision = FUSION_NO_HUMAN;
}

strncpy(f.decisionText, decisionName(f.decision), sizeof(f.decisionText) - 1);

// Propagate radar safety warning to the fusion result for display
f.warning[0] = '\0';
if (rt.radar.compromisedBySafety) {
  strncpy(f.warning, "Radar compromised by close-range safety event", sizeof(f.warning) - 1);
}

snprintf(f.debug, sizeof(f.debug),
  "r=%.2f i=%.2f p=%.2f final=%.2f wr=%.2f wi=%.2f wp=%.2f",
  radar, ir, pos, f.overallConfidence,
  f.radarWeight, f.irWeight, f.positionWeight);
```

```
  f.completed = true;
}

// ---------------------------------------------------------------------------
// beginScan — initialize and start a new scan from the current moment.
// Resets all accumulated results. Safe to call repeatedly between scans.
// ---------------------------------------------------------------------------
static inline void beginScan(ScanRuntimeState& rt, uint32_t nowMs) {
  resetRuntime(rt);
  rt.active        = true;
  rt.state         = SCAN_INIT;
  rt.previousState = SCAN_IDLE;
  rt.scanStartMs   = nowMs;
  rt.stateStartMs  = nowMs;
  rt.lastUpdateMs  = nowMs;
  rt.lastRenderMs  = nowMs;
  rt.renderDirty   = true;
}

// Request cancellation — SCAN_ABORT will be entered on the next updateScanMode() call
static inline void requestCancel(ScanRuntimeState& rt) {
  rt.cancelRequested = true;
}

// Internal: advance to a new state and reset per-state tracking fields
static inline void setState(ScanRuntimeState& rt, ScanState next, uint32_t nowMs) {
  rt.previousState = rt.state;
  rt.state         = next;
  rt.stateStartMs  = nowMs;
  rt.phaseEntered  = false;  // will trigger enterPhase() on next update
  rt.renderDirty   = true;
}

// Internal: run the one-time entry action for the current phase
// Guards against re-entry via phaseEntered flag
static inline void enterPhase(ScanRuntimeState& rt, const ScanCallbacks& cb, uint32_t nowMs) {
```

```
  if (rt.phaseEntered) return;
  rt.phaseEntered = true;
  switch (rt.state) {
    case SCAN_RADAR:
      resetRadarResult(rt.radar);
      rt.radar.phaseStartMs = nowMs;
      if (cb.enterRadarPhase) cb.enterRadarPhase();
      break;
    case SCAN_IR:
      resetIrResult(rt.ir);
      rt.ir.phaseStartMs = nowMs;
      if (cb.enterIrPhase) cb.enterIrPhase();
      break;
    case SCAN_POSITION:
      resetPositionResult(rt.position);
      rt.position.phaseStartMs = nowMs;
      if (cb.enterPositionPhase) cb.enterPositionPhase();
      break;
    default:
      break;
  }
}

// Internal: run the one-time exit action for the given phase
static inline void exitPhase(ScanState state, const ScanCallbacks& cb) {
  switch (state) {
    case SCAN_RADAR:    if (cb.exitRadarPhase)    cb.exitRadarPhase();    break;
    case SCAN_IR:       if (cb.exitIrPhase)       cb.exitIrPhase();       break;
    case SCAN_POSITION: if (cb.exitPositionPhase) cb.exitPositionPhase(); break;
    default: break;
  }
}

// ---------------------------------------------------------------------------
// updateScanMode — main update function, call every loop() iteration while active.
//
// Drives the state machine, latches sensor inputs into results, and fires render
```

```
// and lifecycle callbacks at the configured intervals. Results are valid to read
// when state reaches SCAN_DONE (rt.summaryLatched == true).
//
// Parameters:
//   rt    : runtime state, passed by reference — modified in place
//   cfg   : configuration (from defaultConfig() or custom)
//   in    : current sensor readings (fill before calling)
//   cb    : optional callbacks (all fields may be nullptr)
//   nowMs : current millis() timestamp
// ----------------------------------------------------------------------------
inline void updateScanMode(
  ScanRuntimeState& rt,
  const ScanConfig& cfg,
  const ScanSensorInputs& in,
  const ScanCallbacks& cb,
  uint32_t nowMs
) {
  if (!rt.active) return;
  rt.lastUpdateMs = nowMs;

  // Handle cancellation — exit current phase cleanly before transitioning
  if (rt.cancelRequested && rt.state != SCAN_ABORT && rt.state != SCAN_DONE) {
    exitPhase(rt.state, cb);
    setState(rt, SCAN_ABORT, nowMs);
  }

  switch (rt.state) {
    case SCAN_IDLE:
      return;  // not started yet — nothing to do

    case SCAN_INIT:
      // Brief settle period — waits for sensors to stabilize before radar phase
      if ((nowMs - rt.stateStartMs) >= cfg.initTimeoutMs) {
        setState(rt, SCAN_RADAR, nowMs);
      }
      break;
```

```
case SCAN_RADAR: {
  enterPhase(rt, cb, nowMs);
  if (cb.updateRadarPhase) cb.updateRadarPhase();
  // Latch current readings — confidence is recomputed every frame
  latchRadarResult(rt, in, cfg, nowMs);
  rt.progressPermille = (uint16_t)min<uint32_t>(1000,
    (rt.radar.phaseElapsedMs * 1000UL) / max<uint32_t>(1, cfg.radarMaxDurationMs));
  // Phase exits only on timeout — radar does not exit early on detection
  // because it needs the full budget to accumulate stable vitals
  if (rt.radar.phaseElapsedMs >= cfg.radarMaxDurationMs) {
    rt.radar.timedOut   = true;
    rt.radar.completed  = true;
    exitPhase(rt.state, cb);
    setState(rt, SCAN_IR, nowMs);
  }
  break;
}

case SCAN_IR: {
  enterPhase(rt, cb, nowMs);
  if (cb.updateIrPhase) cb.updateIrPhase();
  latchIrResult(rt, in, nowMs);
  rt.progressPermille = (uint16_t)min<uint32_t>(1000,
    (rt.ir.phaseElapsedMs * 1000UL) / max<uint32_t>(1, cfg.irMaxDurationMs));
  // Exit early once the thermal detector has any meaningful positive signal,
  // whether that comes from the presence subscore, the binary human flag,
  // or the raw classifier confidence.
  bool irReady = rt.ir.humanLike ||
                 (rt.ir.humanPresenceScore >= cfg.irPossibleThreshold) ||
                 (rt.ir.confidenceScore    >= cfg.irPossibleThreshold) ||
                 (rt.ir.phaseElapsedMs >= cfg.irMaxDurationMs);
  if (irReady) {
    rt.ir.completed = true;
    rt.ir.timedOut  = rt.ir.phaseElapsedMs >= cfg.irMaxDurationMs;
    exitPhase(rt.state, cb);
    // Skip position if both radar and IR are below possible threshold
    // and collectPositionOnNegative is not set
```

```
    if (cfg.collectPositionOnNegative ||
        rt.radar.radarConfidence >= cfg.radarPossibleThreshold ||
        rt.ir.confidenceScore    >= cfg.irPossibleThreshold) {
      setState(rt, SCAN_POSITION, nowMs);
    } else {
      setState(rt, SCAN_FUSION, nowMs);
    }
  }
  break;
}

case SCAN_POSITION: {
  enterPhase(rt, cb, nowMs);
  if (cb.updatePositionPhase) cb.updatePositionPhase();
  latchPositionResult(rt, in, nowMs);
  rt.progressPermille = (uint16_t)min<uint32_t>(1000,
    (rt.position.phaseElapsedMs * 1000UL) / max<uint32_t>(1, cfg.positionMaxDurationMs));
  // Exit as soon as any position source is valid, or on timeout
  bool posReady = rt.position.uwbValid || rt.position.gnssValid || rt.position.insValid ||
                  (rt.position.phaseElapsedMs >= cfg.positionMaxDurationMs);
  if (posReady) {
    rt.position.completed = true;
    rt.position.timedOut  = rt.position.phaseElapsedMs >= cfg.positionMaxDurationMs;
    exitPhase(rt.state, cb);
    setState(rt, SCAN_FUSION, nowMs);
  }
  break;
}

case SCAN_FUSION:
  // fuseResults runs every frame during the hold period so the display
  // can show the confidence values updating in real time
  fuseResults(rt, cfg);
  rt.progressPermille = 1000;
  if ((nowMs - rt.stateStartMs) >= cfg.fusionHoldMs) {
    setState(rt, SCAN_DONE, nowMs);
  }
```

```
      break;

    case SCAN_DONE:
      rt.summaryLatched = true;  // results are stable — safe to read from outside
      break;

    case SCAN_ABORT:
      rt.summaryLatched = true;
      break;
  }

  // Progress bar callback — fires every update during active phases
  if (cb.renderProgressBar &&
      (rt.state == SCAN_RADAR || rt.state == SCAN_IR ||
       rt.state == SCAN_POSITION || rt.state == SCAN_FUSION)) {
    cb.renderProgressBar(rt.state, rt.progressPermille);
  }

  // Render callbacks — rate-limited to summaryRefreshMs, or immediate if dirty
  if ((nowMs - rt.lastRenderMs) >= cfg.summaryRefreshMs || rt.renderDirty) {
    rt.lastRenderMs = nowMs;
    rt.renderDirty  = false;
    switch (rt.state) {
      case SCAN_RADAR:    if (cb.renderRadarPhase)    cb.renderRadarPhase(rt, rt.progressPermille);
break;
      case SCAN_IR:       if (cb.renderIrPhase)       cb.renderIrPhase(rt, rt.progressPermille);       break;
      case SCAN_POSITION: if (cb.renderPositionPhase) cb.renderPositionPhase(rt, rt.progressPermille);
break;
      case SCAN_FUSION:   if (cb.renderFusionPhase)   cb.renderFusionPhase(rt, rt.progressPermille);
break;
      case SCAN_DONE:     if (cb.renderDonePhase)     cb.renderDonePhase(rt);                          break;
      case SCAN_ABORT:    if (cb.renderAbortPhase)    cb.renderAbortPhase(rt);                         break;
      default: break;
    }
  }

  // Serial debug — rate-limited to once per second
```

```
  if (cb.serialDebug && (nowMs - rt.lastDebugMs) >= 1000) {
    rt.lastDebugMs = nowMs;
    cb.serialDebug(rt);
  }
}

// ----------------------------------------------------------------------------
// printDebug — convenience helper to dump the scan state to any Stream.
// Pass Serial or any other stream. Rate-limit calls from the serialDebug callback.
// ----------------------------------------------------------------------------
static inline void printDebug(Stream& stream, const ScanRuntimeState& rt) {
  stream.printf(
    "SCAN state=%s radar=%.2f ir=%.2f pos=%.2f fused=%.2f decision=%s warn=%s\n",
    stateName(rt.state),
    rt.radar.radarConfidence,
    rt.ir.confidenceScore,
    rt.position.positionConfidence,
    rt.fusion.overallConfidence,
    rt.fusion.decisionText,
    rt.fusion.warning[0] ? rt.fusion.warning : "-"
  );
}

}

#endif
```

6. Thermal Human Detector header

```
#ifndef THERMAL_HUMAN_DETECTOR_H
#define THERMAL_HUMAN_DETECTOR_H

#include <Arduino.h>
#include <math.h>
#include <string.h>

//
================================================================
========
```

```
// ThermalHuman — MLX90640 human presence detector
//
// Pipeline overview:
//   1. smoothFrame3x3()       — 3x3 weighted spatial blur to reduce pixel noise
//   2. computeFrameStats()    — scene min/max/mean and percentile thresholds
//   3. updateBackground()     — adaptive per-pixel background model
//   4. segmentBlobs()         — threshold + 8-connected flood fill labeling
//   5. extractFeatures()      — geometry, energy distribution, and thermal stats per blob
//   6. classifyThermalBlob()  — score each blob against 9 human body-part classes
//   7. smoothClassification() — EMA temporal smoothing of scores across frames
//   8. detectHumanThermal()   — top-level entry point; returns true if human present
//
// Coordinate system: pixel (0,0) is top-left, X increases right, Y increases down.
// Temperature values are always in degrees Celsius throughout this file.
//
// ============================================================================

namespace ThermalHuman {
// MLX90640 native resolution — 32 columns x 24 rows = 768 pixels
static constexpr int kWidth = 32;
static constexpr int kHeight = 24;
static constexpr int kPixelCount = kWidth * kHeight;
static constexpr int kClassCount = 10;   // Number of human body-part classes the classifier can output

enum ThermalClassId : uint8_t {
  HEAD_ONLY = 0,        // Small oval blob at top of frame — typically 2-8 pixels wide
  HEAD_SHOULDERS,       // Wider blob with top-heavy energy distribution
  SHOULDERS_TORSO,      // Tall blob tapering from wide shoulders to narrower waist
  LOWER_BODY,           // Bottom-heavy blob — legs, feet, or lower torso
  FULL_BODY,            // Tall standing figure spanning most of the frame height
  SITTING,              // Compact blob with centroid shifted downward — seated posture
  CROUCHING,            // Compact near-square blob — crouched or kneeling
  LAYING_DOWN,          // Wide low blob — aspect ratio > 1.75, horizontal orientation
  UNKNOWN_HUMAN_BLOB,   // Blob passes temperature and compactness checks but doesn't
                        // match any specific body-part shape — used as a fallback
```

```
  NO_HUMAN             // No valid human signature detected in this frame
};

// ---------------------------------------------------------------------
// ThermalConfig — all tunable parameters for the detection pipeline.
// Use indoorDefaults(), outdoorDefaults(), or hybridDefaults() as a
// starting point and override individual fields as needed.
// See the tuning guide at the bottom of this file for a suggested workflow.
// ---------------------------------------------------------------------
struct ThermalConfig {
  // Background model update rates
  float bg_alpha_fast;          // Update rate when pixel is NOT above the hot
threshold
  float bg_alpha_slow;          // Update rate when pixel IS above threshold
  float bg_hold_contrast_c;     // If a pixel exceeds this, background update rate
drops to slow
  float bg_delta_c;             // Minimum (pixel - background) delta for segmentation
threshold
  float percentile_delta_c;     // Requires pixel to exceed p75 + this value prevents
warm backgrounds from generating false blobs across the whole
  // Minimum absolute contrast required in standard (indoor) and outdoor modes
  float min_contrast_c;
  float outdoor_contrast_c;
  float warm_scene_boost;       // Multiplier applied to contrast threshold in warm
outdoor scenes
  // Minimum blob area in pixels — scales with estimated distance near/far
  float min_blob_area_near;
  float min_blob_area_far;
  float max_blob_area;          // Maximum blob area in pixels
  float edge_compactness_min;   // Minimum edge compactness

  // Fill ratio bounds — rejects blobs that are too sparse or too solid
  // fill_ratio = hot_pixels / bounding_box_area
  float fill_ratio_min;
  float fill_ratio_max;

  // Hysteresis thresholds for the presence EMA
  // present_on:  EMA must exceed this to trigger a positive detection
  // present_off: EMA must drop below this to clear the detection
  float present_on;
  float present_off;

  // EMA alpha for the presence score and per-class scores
  // rise_alpha: how quickly detection builds up when human is present
  // fall_alpha: how quickly detection decays when human leaves
  float rise_alpha;
```

```
  float fall_alpha;

  // Distance at which min_blob_area transitions from near to far
  float far_distance_m;

  // Aspect ratio thresholds for body-part classification
  // aspect_ratio = bbox_width / bbox_height
  float laying_aspect_min;    // blobs wider than this are candidates for
LAYING_DOWN
  float seated_aspect_min;    // blobs with this ratio and low centroid are SITTING
  float crouch_aspect_min;    // near-square blobs above this are CROUCHING
  float standing_aspect_max;  // blobs narrower than this (tall) are upright poses

  // Temperature plausibility parameters
  // human_temp_center_c: nominal skin/clothing surface temperature
  // human_temp_variation_f: allowed deviation in Fahrenheit (converted
internally to °C)
  // max_hotspot_temp_c: anything above this is likely a heater/electronics, not
human
  // min_mean_human_temp_c: absolute minimum mean temperature for a valid
human blob
  float human_temp_center_c;
  float human_temp_variation_f;
  float max_hotspot_temp_c;
  float min_mean_human_temp_c;

  // If true, always use outdoor segmentation parameters regardless of scene
temperature
  bool force_outdoor;
};

// ---------------------------------------------------------------------
// ThermalFeatures — geometric and thermal measurements extracted from a
// single blob after segmentation. All fields are populated by extractFeatures().
// These are passed to classifyThermalBlob() and also exposed to the caller
// for display and logging via the debug string.
// ---------------------------------------------------------------------
struct ThermalFeatures {
  // Bounding box in pixel coordinates (top-left origin)
  int bbox_x, bbox_y, bbox_w, bbox_h;

  // Centroid of the hot pixel mass (weighted by pixel membership, not
temperature)
  float centroid_x, centroid_y;

  // Alias for bbox_w/bbox_h — kept for clarity in shape checks
```

```
  float blob_width, blob_height;

  // Width / height ratio of the bounding box
  float aspect_ratio;

  // Total number of pixels classified as hot (above segmentation threshold)
  int hot_pixel_count;

  // Mean and maximum temperature of hot pixels in °C
  float mean_target_temp, max_target_temp;

  // Mean contrast of hot pixels above their local background estimate
  float contrast_above_local_background;

  // Final confidence output [0,1] after temporal smoothing — mirrors
ThermalScores.confidence
  float confidence;

  // Mean temperature of pixels in the border region around the blob
  // Used as a local reference to avoid over-relying on the global background
model
  float local_background;

  // Temperature threshold used by segmentation for this blob's location
  float threshold_c;

  // hot_pixel_count / bbox_area
  float fill_ratio;

  // Alias for fill_ratio — used in classification scoring
  float compactness;

  // hot_pixel_count / perimeter_pixels — measures how solid vs. hollow the blob
is
  // Low values indicate a ring or hollow shape (unlikely to be human)
  float edge_compactness;

  // Temporal stability metrics updated by smoothClassification()
  float temporal_persistence;  // fraction of recent frames with a stable blob
  float stability;             // 1 - (centroid jump / 8 pixels) clamped to [0,1]

  // Penalty applied when blob is smaller than the distance-scaled minimum area
  float size_penalty;

  // Distance estimate passed in from external ranging (UWB/radar/USS) — NAN
if unknown
```

```
  float distance_estimate_m;

  // ---------------------------------------------------------------------
  // Energy distribution features — the blob is divided into thirds and halves
  // vertically. Each region's contribution is expressed as a fraction of total
  // blob energy (temperature above local background). Used extensively in
  // body-part classification to distinguish head-heavy vs. leg-heavy poses.
  // ---------------------------------------------------------------------
  float top_half_energy;        // fraction of energy in top 50% of bbox height
  float bottom_half_energy;     // fraction of energy in bottom 50% of bbox height
  float top_third_energy;       // fraction of energy in top 33% of bbox height
  float middle_third_energy;    // fraction of energy in middle 33%
  float lower_third_energy;     // fraction of energy in bottom 33%

  // Width of the hot pixel cluster in each vertical third
  // Used to detect tapering shapes like shoulders-to-waist or head-to-shoulders
  float top_third_width;
  float middle_third_width;
  float lower_third_width;

  // top_half_energy - bottom_half_energy — positive means top-heavy (upright
human)
  float vertical_gradient;

  // Alias for aspect_ratio — retained for readability in shape scoring
  float horizontal_spread;

  // Score [0,1] measuring how well the blob tapers from middle outward
  // High score = middle is widest (torso-like); low score = uniform or inverted
taper
  float width_taper_score;

  // Reserved for future left-right symmetry scoring (not yet implemented)
  float vertical_symmetry;

  // Pixel coordinates of the hottest pixel in the blob
  // Used to verify that the hottest point is near the top (head region)
  int hottest_x, hottest_y;

  // Null-terminated debug string formatted by extractFeatures() — contains
  // all key metrics in a single line for Serial/log output
  char debug[192];
};

// ---------------------------------------------------------------------
// ThermalScores — classification output for a single blob.
```

```
// All scores are in [0,1]. best_class is the highest-scoring human class.
// ---------------------------------------------------------------------
struct ThermalScores {
  // Overall probability that this blob contains a human [0,1]
  // Drives the hysteresis EMA — crosses present_on to trigger detection
  float human_presence_score;
  // Per-class scores — index matches ThermalClassId enum values
  float class_scores[kClassCount];
  // Combined confidence after temporal smoothing — accounts for stability
  // and persistence in addition to the instantaneous classification score
  float confidence;
  // The class with the highest score among the 9 human classes
  // Set to NO_HUMAN if no class exceeds the detection threshold
  ThermalClassId best_class;
};

// ---------------------------------------------------------------------
// ThermalState — per-instance mutable state that persists across frames.
// One ThermalState should be allocated per sensor and initialized with
// initState() before the first detectHumanThermal() call.
// ---------------------------------------------------------------------
struct ThermalState {
   // Per-pixel adaptive background model
  float background[kPixelCount];
  float background_var[kPixelCount];
  float smoothed[kPixelCount];
  float contrast[kPixelCount];
  uint8_t mask[kPixelCount];
  int16_t labels[kPixelCount];
  uint8_t visited[kPixelCount];
  int stack[kPixelCount];

  bool background_ready;
  bool auto_outdoor;
  float presence_ema;           // Exponential moving average of
human_presence_score across frames
  float class_ema[kClassCount];  // EMA of each class score across frames
  float prev_centroid_x;
  float prev_centroid_y;
  uint8_t stable_frames;
};

// ---------------------------------------------------------------------
// Inline math
// ---------------------------------------------------------------------
```

```
// Clamp v to [0, 1]
static inline float clamp01(float v) {
  if (v < 0.0f) return 0.0f;
  if (v > 1.0f) return 1.0f;
  return v;
}

// Linear interpolation between a and b by factor t in [0,1]
static inline float lerp(float a, float b, float t) {
  return a + (b - a) * t;
}

// Returns 0 if v <= lo, 1 if v >= hi, linear ramp between.
// Used throughout classification to smoothly score features against thresholds
// instead of hard binary checks that would cause score discontinuities.
static inline float scoreBand(float v, float lo, float hi) {
  if (v <= lo) return 0.0f;
  if (v >= hi) return 1.0f;
  return (v - lo) / (hi - lo);
}

// Returns the human-readable name string for a class ID.
// Used in debug output and the LCD HUD display.
static inline const char* className(ThermalClassId id) {
  switch (id) {
    case HEAD_ONLY:          return "head_only";
    case HEAD_SHOULDERS:     return "head_shoulders";
    case SHOULDERS_TORSO:    return "shoulders_torso";
    case LOWER_BODY:         return "lower_body";
    case FULL_BODY:          return "full_body";
    case SITTING:            return "sitting";
    case CROUCHING:          return "crouching";
    case LAYING_DOWN:        return "laying_down";
    case UNKNOWN_HUMAN_BLOB: return "unknown_human_blob";
    default: return "no_human";
  }
}

// ---------------------------------------------------------------------
// Default configuration presets
// indoorDefaults()  — standard indoor operation, lower thresholds
// outdoorDefaults() — higher thresholds for warm/sunny outdoor scenes
// hybridDefaults()  — middle ground, works well for doorway/transition zones
// ---------------------------------------------------------------------
static inline ThermalConfig indoorDefaults() {
  ThermalConfig c{};
```

```
  c.bg_alpha_fast           = 0.020f;  // background updates at ~2% per frame when
cool
  c.bg_alpha_slow           = 0.003f;  // drops to 0.3% when pixel is above hot
threshold
  c.bg_hold_contrast_c      = 0.85f;   // above this contrast, slow down
background update
  c.bg_delta_c              = 0.55f;   // minimum pixel-background delta to flag as hot
  c.percentile_delta_c      = 0.80f;   // pixel must also exceed p75 + this value
  c.min_contrast_c          = 1.00f;   // minimum absolute contrast for indoor
segmentation
  c.outdoor_contrast_c      = 1.30f;   // contrast floor when outdoor mode activates
  c.warm_scene_boost        = 1.35f;   // contrast multiplier in warm outdoor
scenes
  c.min_blob_area_near      = 8.0f;    // minimum pixels for a valid blob at close
range
  c.min_blob_area_far       = 3.0f;    // minimum pixels for a valid blob at far range
  c.max_blob_area           = 340.0f;  // reject blobs covering more than ~44% of
frame
  c.edge_compactness_min    = 0.12f;   // reject hollow/ring blobs below this ratio
  c.fill_ratio_min          = 0.14f;   // reject sparse blobs (too many holes)
  c.fill_ratio_max          = 0.90f;   // reject overly solid blobs (likely non-human
surface)
  c.present_on              = 0.66f;   // EMA must exceed this to trigger positive
detection
  c.present_off             = 0.50f;   // EMA must drop below this to clear detection
  c.rise_alpha              = 0.12f;   // presence EMA rises at 12% per frame
  c.fall_alpha              = 0.04f;   // presence EMA falls at 4% per frame (slower
decay)
  c.far_distance_m          = 4.0f;    // distance at which blob area transitions to far
minimum
  c.laying_aspect_min       = 1.75f;   // width/height ratio above which blob is
considered laying
  c.seated_aspect_min       = 0.75f;   // minimum ratio for seated classification
  c.crouch_aspect_min       = 0.95f;   // minimum ratio for crouching classification
  c.standing_aspect_max     = 1.35f;   // maximum ratio for upright standing
poses
  c.human_temp_center_c     = 33.2f;   // nominal skin/clothing surface
temperature
  c.human_temp_variation_f  = 12.0f;   // ±12°F (~6.7°C) variation from center is
accepted
  c.max_hotspot_temp_c      = 45.0f;   // above this temperature, likely a
heater/device
  c.min_mean_human_temp_c   = (78.0f - 32.0f) * (5.0f / 9.0f);  // ~25.6°C
minimum mean
  c.force_outdoor           = false;
  return c;
```

```
}

static inline ThermalConfig outdoorDefaults() {
  // Start from indoor defaults and tighten thresholds for warm outdoor conditions.
  // Higher bg_delta and contrast values compensate for warm sun-heated
surfaces
  // that would otherwise flood the segmentation mask with false positives.
  ThermalConfig c = indoorDefaults();
  c.bg_alpha_fast     = 0.028f;  // background adapts slightly faster outdoors
  c.bg_alpha_slow     = 0.006f;
  c.bg_delta_c        = 0.68f;   // higher threshold — outdoor surfaces run warmer
  c.percentile_delta_c = 0.92f;
  c.min_contrast_c    = 1.10f;
  c.outdoor_contrast_c = 1.35f;
  c.warm_scene_boost  = 1.45f;
  c.min_blob_area_near = 7.0f;
  c.min_blob_area_far  = 2.0f;
  c.force_outdoor     = true;    // lock to outdoor mode regardless of scene
temperature
  return c;
}

static inline ThermalConfig hybridDefaults() {
  // Balanced preset for environments that transition between indoor and outdoor
  // (doorways, semi-covered areas, vehicle interiors with sun exposure).
  // Stricter presence thresholds than indoor to reduce false positives from
  // mixed thermal backgrounds, but softer than pure outdoor for sensitivity.
  ThermalConfig c = indoorDefaults();
  c.bg_delta_c         = 0.60f;
  c.percentile_delta_c = 0.90f;
  c.min_contrast_c     = 1.10f;
  c.outdoor_contrast_c = 1.45f;
  c.warm_scene_boost   = 1.45f;
  c.min_blob_area_near = 9.0f;   // slightly stricter than indoor to reduce warm-
surface hits
  c.min_blob_area_far  = 2.0f;
  c.edge_compactness_min = 0.12f;
  c.fill_ratio_min     = 0.14f;
  c.fill_ratio_max     = 0.86f;
  c.present_on         = 0.76f;  // higher on-threshold for tighter false-positive
control
  c.present_off        = 0.60f;
  c.rise_alpha         = 0.10f;
  c.fall_alpha         = 0.04f;
  return c;
}
```

```
// Reset all state to zero and seed centroid history to an invalid value
// so the first stability calculation doesn't use uninitialized memory.
// Must be called once before the first detectHumanThermal() call.
static inline void initState(ThermalState& state) {
  memset(&state, 0, sizeof(state));
  state.prev_centroid_x = -1000.0f;  // sentinel — far outside any valid pixel
coordinate
  state.prev_centroid_y = -1000.0f;
}

// ---------------------------------------------------------------------
// smoothFrame3x3 — weighted 3x3 spatial blur
//
// Kernel weights:
//   corner neighbors: 0.75
//   cardinal neighbors: 1.25
//   center pixel: 2.5
//
// This is a non-uniform kernel that preserves the center pixel more strongly
// than a uniform box blur, reducing noise while retaining blob edges.
// Edges and corners are handled by clamping to the nearest valid pixel.
// ---------------------------------------------------------------------
static inline void smoothFrame3x3(const float* src, float* dst) {
  for (int y = 0; y < kHeight; y++) {
    for (int x = 0; x < kWidth; x++) {
      float sum = 0.0f;
      float wsum = 0.0f;
      for (int oy = -1; oy <= 1; oy++) {
        int yy = constrain(y + oy, 0, kHeight - 1);
        for (int ox = -1; ox <= 1; ox++) {
          int xx = constrain(x + ox, 0, kWidth - 1);
          // Center gets 2.5x weight, cardinal neighbors 1.25x, diagonals 0.75x
          float w = (ox == 0 && oy == 0) ? 2.5f : ((ox == 0 || oy == 0) ? 1.25f : 0.75f);
          sum += src[yy * kWidth + xx] * w;
          wsum += w;
        }
      }
      dst[y * kWidth + x] = sum / wsum;
    }
  }
}

// ---------------------------------------------------------------------
// computeFrameStats — single-pass scene statistics
//
```

```
// Computes min, max, and mean temperature, then builds a 64-bin histogram
// to estimate the 75th and 90th percentiles. These percentiles are used
// as scene-adaptive baselines for the segmentation threshold so that the
// detector adjusts automatically to different ambient temperatures without
// requiring manual calibration.
// ------------------------------------------------------------------------
static inline void computeFrameStats(const float* frame, float& min_t, float&
max_t, float& mean_t, float& p75, float& p90) {
  min_t = 1000.0f;
  max_t = -1000.0f;
  mean_t = 0.0f;

  // Single pass to find extremes and accumulate mean
  for (int i = 0; i < kPixelCount; i++) {
    float t = frame[i];
    if (t < min_t) min_t = t;
    if (t > max_t) max_t = t;
    mean_t += t;
  }
  mean_t /= kPixelCount;

  // Build 64-bin histogram over the full temperature range
  int hist[64];
  memset(hist, 0, sizeof(hist));
  float span = max(0.5f, max_t - min_t);    // guard against degenerate frames
  for (int i = 0; i < kPixelCount; i++) {
    int b = (int)(((frame[i] - min_t) / span) * 63.0f);
    b = constrain(b, 0, 63);
    hist[b]++;
  }

  // Walk histogram to find temperature at a given percentile rank
  auto percentile = [&](int target) {
    int acc = 0;
    for (int i = 0; i < 64; i++) {
      acc += hist[i];
      if (acc >= target) return min_t + span * ((float)i / 63.0f);
    }
    return max_t;
  };

  p75 = percentile((int)(kPixelCount * 0.75f));
  p90 = percentile((int)(kPixelCount * 0.90f));
}

// ------------------------------------------------------------------------
```

```
// minBlobAreaForDistance — distance-adaptive minimum blob size
//
// At close range a human fills many pixels; at far range the same human
// may only cover 2-3 pixels. This linearly interpolates between the near
// and far area minimums based on the estimated subject distance.
// ----------------------------------------------------------------------
static inline float minBlobAreaForDistance(const ThermalConfig& cfg, float
distance_m) {
  if (!isfinite(distance_m) || distance_m <= 0.5f) return cfg.min_blob_area_near;
  float t = clamp01((distance_m - 0.5f) / max(0.5f, cfg.far_distance_m - 0.5f));
  return lerp(cfg.min_blob_area_near, cfg.min_blob_area_far, t);
}
// ----------------------------------------------------------------------
// updateBackground — per-pixel adaptive background model
//
// On the first call, seeds the background with the current frame directly.
// On subsequent calls, updates each pixel with a slow EMA:
//   - If the pixel is well above the hot threshold (bg_hold_contrast_c),
//     the update rate drops to bg_alpha_slow so a stationary warm subject
//     doesn't get absorbed into the background over time.
//   - Otherwise the pixel updates at bg_alpha_fast to track ambient changes.
//
// background_var tracks per-pixel noise level and is updated as the
// mean absolute deviation from the background estimate.
// ----------------------------------------------------------------------}
inline void updateBackground(
  ThermalState& state,
  const float* frame,
  uint32_t timestamp_ms,
  float sensor_ambient_c,
  const ThermalConfig& cfg
){
  (void)timestamp_ms;   // reserved for future time-gated background logic

  if (!state.background_ready) {
    // Seed background from first frame — variance starts at a small positive value
    for (int i = 0; i < kPixelCount; i++) {
      state.background[i] = frame[i];
      state.background_var[i] = 0.15f;
    }
    state.background_ready = true;
    return;
  }

  // Scale bg_alpha fast with INS ambient temp
  float alpha_fast = cfg.bg_alpha_fast;
```

```
  if (isfinite(sensor_ambient_c)) {
    float warmBias = clamp01((sensor_ambient_c - 15.0f)/20.0f);
    alpha_fast = lerp(cfg.bg_alpha_fast, cfg.bg_alpha_fast * 1.6f, warmBias);
  }

  for (int i = 0; i < kPixelCount; i++) {
    float diff = frame[i] - state.background[i];

    // Choose update rate: freeze background under hot pixels to prevent subject
absorption
    float alpha = (diff > cfg.bg_hold_contrast_c) ? 0.0f :
              (diff > cfg.bg_hold_contrast_c * 0.5f) ? cfg.bg_alpha_slow :
cfg.bg_alpha_fast;
    state.background[i] += alpha * diff;

    // Track noise: variance EMA converges toward mean absolute deviation
    state.background_var[i] += 0.08f * (fabsf(diff) - state.background_var[i]);
    if (state.background_var[i] < 0.05f) state.background_var[i] = 0.05f;  // floor to
prevent zero
  }
}

// ---------------------------------------------------------------------
// segmentBlobs — threshold segmentation and 8-connected flood fill
//
// Each pixel is flagged as hot if it exceeds both:
//   1. background[i] + bg_delta_c  (local adaptive threshold)
//   2. p75 + percentile_delta_c    (global scene percentile threshold)
//   3. p75                         (absolute scene floor)
//   AND the computed contrast exceeds the mode-appropriate floor
//
// Hot pixels are then grouped using an iterative 8-connected flood fill
// (explicit stack to avoid recursion overflow on embedded targets).
// Blobs outside the area bounds [min_area, max_blob_area] are discarded
// and their labels set back to -1.
//
// Returns: number of valid blobs found (0 if none pass area filters)
// ---------------------------------------------------------------------
static inline int segmentBlobs(
  ThermalState& state,
  const float* frame,
  float p75,
  float p90,
  float sensor_ambient_c,
  float distance_m,
  const ThermalConfig& cfg
```

```
) {
  memset(state.mask, 0, sizeof(state.mask));
  memset(state.visited, 0, sizeof(state.visited));
  for (int i = 0; i < kPixelCount; i++) state.labels[i] = -1;

 // Outdoor mode is latched in detectHumanThermal() — use it directly here
  state.auto_outdoor = cfg.force_outdoor || state.auto_outdoor;
  float contrast_floor = state.auto_outdoor ? cfg.outdoor_contrast_c :
cfg.min_contrast_c;
  float dynamic_contrast = contrast_floor * (state.auto_outdoor ?
cfg.warm_scene_boost : 1.0f);

  // Threshold each pixel against both the adaptive background and the scene
percentile
  for (int i = 0; i < kPixelCount; i++) {
    float bg = state.background[i];
    float threshold = max(bg + cfg.bg_delta_c, p75 + cfg.percentile_delta_c);
    state.contrast[i] = frame[i] - bg;
    bool hot =
      frame[i] > threshold &&
      state.contrast[i] > dynamic_contrast &&
      frame[i] > p75;
    state.mask[i] = hot ? 1 : 0;
  }

  // Flood fill each unvisited hot pixel to find connected blobs
  int label = 0;
  float min_area = minBlobAreaForDistance(cfg, distance_m);
  for (int i = 0; i < kPixelCount; i++) {
    if (!state.mask[i] || state.visited[i]) continue;

    // BFS/DFS flood fill using explicit stack — 8-connected neighborhood
    int sp = 0;
    int area = 0;
    state.stack[sp++] = i;
    state.visited[i] = 1;

    while (sp > 0) {
      int idx = state.stack[--sp];
      state.labels[idx] = label;
      area++;
      int x = idx % kWidth;
      int y = idx / kWidth;
      const int n8[8][2] = {{1,0},{-1,0},{0,1},{0,-1},{1,1},{1,-1},{-1,1},{-1,-1}};
      for (int n = 0; n < 8; n++) {
        int nx = x + n8[n][0];
```

```
        int ny = y + n8[n][1];
        if (nx < 0 || nx >= kWidth || ny < 0 || ny >= kHeight) continue;
        int ni = ny * kWidth + nx;
        if (!state.mask[ni] || state.visited[ni]) continue;
        state.visited[ni] = 1;
        state.stack[sp++] = ni;
      }
    }
    // Reject blobs outside the valid area range — clear their labels
    if (area < min_area || area > cfg.max_blob_area) {
      for (int j = 0; j < kPixelCount; j++) {
        if (state.labels[j] == label) state.labels[j] = -1;
      }
    } else {
      label++;     // only increment label counter for blobs that passed area check
    }
  }
  return label;  // total number of valid blobs
}

// ---------------------------------------------------------------------
// extractFeatures — compute all geometric and thermal features for one blob
//
// Makes two passes over the labeled pixels:
//   Pass 1: bounding box, centroid, temperature stats, perimeter count
//   Pass 2: vertical energy thirds, width per third, hottest pixel location
//
// Also samples the border region outside the bounding box to estimate
// local background temperature independently of the global model.
//
// label: the blob index to extract (from segmentBlobs return value - 1 downward)
// p75:   scene 75th percentile temperature — used as fallback local background
// ---------------------------------------------------------------------
inline void extractFeatures(
  const ThermalState& state,
  const float* frame,
  int label,
  float p75,
  float distance_m,
  const ThermalConfig& cfg,
  ThermalFeatures& out
) {
  memset(&out, 0, sizeof(out));
  out.bbox_x = kWidth;
  out.bbox_y = kHeight;
  out.distance_estimate_m = distance_m;
```

```
  out.max_target_temp = -1000.0f;

  float sum_t = 0.0f;
  float sum_c = 0.0f;
  float sum_x = 0.0f;
  float sum_y = 0.0f;
  int max_x = 0;
  int max_y = 0;
  int perimeter = 0;

  // Pass 1: bounding box, centroid, temperature stats, perimeter
  for (int i = 0; i < kPixelCount; i++) {
    if (state.labels[i] != label) continue;
    int x = i % kWidth;
    int y = i / kWidth;
    float temp = frame[i];
    float contrast = state.contrast[i];
    out.hot_pixel_count++;
    sum_t += temp;
    sum_c += contrast;
    sum_x += x;
    sum_y += y;
    if (x < out.bbox_x) out.bbox_x = x;
    if (x > max_x) max_x = x;
    if (y < out.bbox_y) out.bbox_y = y;
    if (y > max_y) max_y = y;
    if (temp > out.max_target_temp) {
      out.max_target_temp = temp;
      out.hottest_x = x;
      out.hottest_y = y;
    }
    // A pixel is a perimeter pixel if any of its 4-connected neighbors is
    // outside the blob — used to compute edge_compactness
    bool edge = false;
    const int n4[4][2] = {{1,0},{-1,0},{0,1},{0,-1}};
    for (int n = 0; n < 4; n++) {
      int nx = x + n4[n][0];
      int ny = y + n4[n][1];
      if (nx < 0 || nx >= kWidth || ny < 0 || ny >= kHeight || state.labels[ny * kWidth +
nx] != label) edge = true;
    }
    if (edge) perimeter++;
  }

  if (out.hot_pixel_count <= 0) return;
  // Derive bounding box dimensions and shape metrics
```

```
  out.bbox_w = max_x - out.bbox_x + 1;
  out.bbox_h = max_y - out.bbox_y + 1;
  out.blob_width = out.bbox_w;
  out.blob_height = out.bbox_h;
  out.aspect_ratio = (out.bbox_h > 0) ? (float)out.bbox_w / (float)out.bbox_h :
0.0f;
  out.centroid_x = sum_x / out.hot_pixel_count;
  out.centroid_y = sum_y / out.hot_pixel_count;
  out.mean_target_temp = sum_t / out.hot_pixel_count;
  out.contrast_above_local_background = sum_c / out.hot_pixel_count;

  int bbox_area = max(1, out.bbox_w * out.bbox_h);
  out.fill_ratio = (float)out.hot_pixel_count / bbox_area;
  out.compactness = out.fill_ratio;
  out.edge_compactness = (float)out.hot_pixel_count / max(1, perimeter);

  // Sample the 1-pixel border around the bounding box for local background
estimate
  // This avoids the blob's own hot pixels biasing the local temperature reference
  float bg_sum = 0.0f;
  int bg_count = 0;
  for (int y = max(0, out.bbox_y - 1); y <= min(kHeight - 1, out.bbox_y +
out.bbox_h); y++) {
    for (int x = max(0, out.bbox_x - 1); x <= min(kWidth - 1, out.bbox_x +
out.bbox_w); x++) {
      int idx = y * kWidth + x;
      if (state.labels[idx] == label) continue;
      bg_sum += frame[idx];
      bg_count++;
    }
  }
  out.local_background = (bg_count > 0) ? (bg_sum / bg_count) : p75;
  out.threshold_c = max(out.local_background + cfg.bg_delta_c, p75 +
cfg.percentile_delta_c);

  // Pass 2: vertical energy distribution — divide blob into thirds and halves
  // by bounding box height and accumulate energy in each region
  float top = 0.0f, bottom = 0.0f, t1 = 0.0f, t2 = 0.0f, t3 = 0.0f;
  int split_half = out.bbox_y + out.bbox_h / 2;
  int split1 = out.bbox_y + max(1, out.bbox_h / 3);
  int split2 = out.bbox_y + max(2, (2 * out.bbox_h) / 3);
  // Track width of hot pixels in each vertical third for taper detection
  int top_min = kWidth, top_max = 0, mid_min = kWidth, mid_max = 0, low_min =
kWidth, low_max = 0;
  int top_n = 0, mid_n = 0, low_n = 0;
```

```
  for (int y = out.bbox_y; y < out.bbox_y + out.bbox_h; y++) {
    for (int x = out.bbox_x; x < out.bbox_x + out.bbox_w; x++) {
      int idx = y * kWidth + x;
      if (state.labels[idx] != label) continue;
      float c = frame[idx] - out.local_background;  // energy relative to local
background
      if (y < split_half) top += c; else bottom += c;
      if (y < split1) {
        t1 += c; top_n++;
        if (x < top_min) top_min = x;
        if (x > top_max) top_max = x;
      } else if (y < split2) {
        t2 += c; mid_n++;
        if (x < mid_min) mid_min = x;
        if (x > mid_max) mid_max = x;
      } else {
        t3 += c; low_n++;
        if (x < low_min) low_min = x;
        if (x > low_max) low_max = x;
      }
    }
  }
  // Normalize energy fractions so they sum to approximately 1.0
  float total = max(0.001f, t1 + t2 + t3);
  out.top_half_energy = top / max(0.001f, top + bottom);
  out.bottom_half_energy = bottom / max(0.001f, top + bottom);
  out.top_third_energy = t1 / total;
  out.middle_third_energy = t2 / total;
  out.lower_third_energy = t3 / total;
  // Width of the hot pixel cluster in each third — used to detect taper
  out.top_third_width = top_n ? (top_max - top_min + 1) : 0.0f;
  out.middle_third_width = mid_n ? (mid_max - mid_min + 1) : 0.0f;
  out.lower_third_width = low_n ? (low_max - low_min + 1) : 0.0f;
  // Width taper score — how closely the blob matches a torso-like shape
  // where the middle third is widest and both top and bottom taper inward.
  // Score = 0 if middle is narrow; score = 1 if top ~80% and bottom ~90% of
middle width
  out.width_taper_score = 0.0f;
if (out.middle_third_width > 0.5f) {
  float top_to_mid = out.top_third_width / max(1.0f, out.middle_third_width);
  float low_to_mid = out.lower_third_width / max(1.0f, out.middle_third_width);

  out.width_taper_score =
    clamp01(1.0f - fabsf(top_to_mid - 0.8f)) * 0.5f +
    clamp01(1.0f - fabsf(low_to_mid - 0.9f)) * 0.5f;
}
```

```
// Vertical gradient
out.vertical_gradient = out.top_half_energy - out.bottom_half_energy;
out.horizontal_spread = out.aspect_ratio;
  // Stability: measures how far the centroid moved from last frame
  // Clamps to [0,1] where 1 = no movement, 0 = moved 8+ pixels
  float centroid_jump = sqrtf(
    (out.centroid_x - state.prev_centroid_x) * (out.centroid_x -
state.prev_centroid_x) +
    (out.centroid_y - state.prev_centroid_y) * (out.centroid_y -
state.prev_centroid_y)
  );
  out.stability = clamp01(1.0f - centroid_jump / 8.0f);
  // Temporal persistence: fraction of the last ~8 frames that had a stable blob
  out.temporal_persistence = clamp01((float)state.stable_frames / 8.0f);
  // Size penalty: how far below the minimum area threshold this blob falls
  // 0 = blob is at or above minimum; 1 = blob has zero pixels
  float min_area = minBlobAreaForDistance(cfg, distance_m);
  out.size_penalty = clamp01((min_area - out.hot_pixel_count) / max(1.0f,
min_area));
  // LCD display
  snprintf(
    out.debug, sizeof(out.debug),
    "bbox=%d,%d,%d,%d area=%d asp=%.2f fill=%.2f dT=%.2f top=%.2f
mid=%.2f low=%.2f stab=%.2f pers=%.2f",
    out.bbox_x, out.bbox_y, out.bbox_w, out.bbox_h, out.hot_pixel_count,
out.aspect_ratio,
    out.fill_ratio, out.contrast_above_local_background, out.top_third_energy,
    out.middle_third_energy, out.lower_third_energy, out.stability,
out.temporal_persistence
  );
}

// ---------------------------------------------------------------------
// classifyThermalBlob — score a blob against all 9 human body-part classes
//
// Each class score is a weighted combination of shape, energy distribution,
// and temperature features. All inputs are normalized to [0,1] via scoreBand()
// before weighting so that no single feature can dominate the sum.
//
// After scoring, non-human artifacts are penalized (tinyHotBlobPenalty,
// overCompactPenalty, flatHotBarPenalty, singleHotspotPenalty) and
// temperature plausibility scales all scores multiplicatively.
//
// The final human_presence_score drives the EMA in smoothClassification().
// ---------------------------------------------------------------------
```

```
inline void classifyThermalBlob(const ThermalFeatures& f, const ThermalConfig&
cfg, ThermalScores& s) {
  memset(&s, 0, sizeof(s));
  s.best_class = NO_HUMAN;

  // Hard reject: temperatures above ~40°C are heaters, electronics, or hot
surfaces.
  // Rejecting early prevents these blobs from suppressing valid cooler human
blobs
  // that might exist elsewhere in the frame.
  const float hardHumanMaxTargetC = (104.0f - 32.0f) * (5.0f / 9.0f);
  if (isfinite(f.max_target_temp) && f.max_target_temp > hardHumanMaxTargetC)
{
    s.class_scores[NO_HUMAN] = 1.0f;
    s.confidence = 0.0f;
    s.human_presence_score = 0.0f;
    return;
  }
  // Compute temperature plausibility inputs
  const float humanTempVariationC = cfg.human_temp_variation_f * (5.0f / 9.0f);
  const float humanTempMinC = cfg.human_temp_center_c -
humanTempVariationC;
  const float humanTempMaxC = cfg.human_temp_center_c +
humanTempVariationC;
  // How close the mean temperature is to the nominal human surface
temperature
  float meanTempScore =
    1.0f - clamp01(fabsf(f.mean_target_temp - cfg.human_temp_center_c) /
    max(0.25f, humanTempVariationC));
  // How close the peak temperature is to the nominal center
  float maxTempScore =
    1.0f - clamp01(fabsf(f.max_target_temp - cfg.human_temp_center_c) /
    max(0.25f, humanTempVariationC + 0.75f));
  // Ramp score from 0 to 1 as mean temperature crosses the minimum human
floor
  float meanFloorScore =
    scoreBand(f.mean_target_temp, cfg.min_mean_human_temp_c - 1.5f,
              cfg.min_mean_human_temp_c + 2.0f);
  // Wider band score for mean temperature — tolerates more clothing variation
  float meanBandScore =
    1.0f - clamp01(fabsf(f.mean_target_temp - cfg.human_temp_center_c) /
    max(0.5f, humanTempVariationC + 1.5f));
  // Wider band score for peak temperature
  float maxBandScore =
    1.0f - clamp01(fabsf(f.max_target_temp - cfg.human_temp_center_c) /
    max(0.5f, humanTempVariationC + 3.0f));
```

```
  // Penalty for hotspot temperatures above the human maximum
  float hotspotPenalty = 0.0f;
  if (isfinite(f.max_target_temp) && f.max_target_temp >
cfg.max_hotspot_temp_c) {
    hotspotPenalty = clamp01((f.max_target_temp - cfg.max_hotspot_temp_c) /
8.0f);
  }

  // Combined temperature plausibility score [0,1]
  // Weighted combination of floor (is it warm enough?), band (is it in the right
range?),
  // and peak checks, minus a penalty for overly hot spots
  float temperaturePlausibility =
    clamp01(
      0.45f * meanFloorScore +
      0.35f * meanBandScore +
      0.20f * maxBandScore -
      0.35f * hotspotPenalty
    );
  // Hottest pixel location features
  float hottest_near_top = (f.hottest_y <= (f.bbox_y + max(1, f.bbox_h / 3))) ?
1.0f : 0.0f;
  float hottest_near_center =
    clamp01((0.5f * f.bbox_w - fabsf(f.hottest_x - (f.bbox_x + 0.5f * f.bbox_w))) /
          (0.5f * f.bbox_w + 0.1f));
  // Boolean temperature band check — mean and peak must both fall within the
  // human plausibility range for the blob to pass final detection
  bool tempBandOk =
    isfinite(f.mean_target_temp) &&
    isfinite(f.max_target_temp) &&
    f.mean_target_temp >= (cfg.min_mean_human_temp_c - 1.5f) &&
    f.mean_target_temp <= (humanTempMaxC + 1.5f) &&
    f.max_target_temp >= (humanTempMinC - 1.0f) &&
    f.max_target_temp <= (cfg.max_hotspot_temp_c + 1.5f);

  // Shape classification booleans — used to gate class-specific score boosts
  // and to determine which fallback path is used when no specific class matches

  // Head: small oval blob with aspect ratio near 1.0
  bool headOvalShape =
    f.bbox_w >= 2 && f.bbox_w <= 8 &&
    f.bbox_h >= 2 && f.bbox_h <= 9 &&
    f.aspect_ratio >= 0.55f && f.aspect_ratio <= 1.55f;

  // Head at far field: very few pixels but still oval and at distance > 3m
  bool headFarField =
```

```
    f.hot_pixel_count >= 2 && f.hot_pixel_count <= 6 &&
    f.fill_ratio >= 0.40f &&
    f.aspect_ratio >= 0.50f && f.aspect_ratio <= 2.0f &&
    isfinite(f.distance_estimate_m) && f.distance_estimate_m >= 3.0f;

  headOvalShape = headOvalShape || headFarField;

  // Head+Shoulders: wider top than bottom with moderate height
  bool headShouldersShape =
    f.bbox_w >= 3 && f.bbox_w <= 14 &&
    f.bbox_h >= 3 && f.bbox_h <= 12 &&
    f.top_third_width >= max(1.0f, f.middle_third_width * 0.70f) &&
    f.middle_third_width >= max(1.0f, f.lower_third_width * 0.85f);

  // Torso+Legs: tall blob with roughly consistent width from top to bottom
  bool torsoLegsShape =
    f.bbox_h >= 5 &&
    f.aspect_ratio >= 0.35f && f.aspect_ratio <= 1.25f &&
    f.middle_third_width >= max(1.0f, f.top_third_width * 0.85f) &&
    f.lower_third_width <= max(1.0f, f.middle_third_width * 1.15f);

  // Full body: combines head oval at top with torso/legs below
  bool fullBodyShape =
    headOvalShape &&
    torsoLegsShape &&
    f.top_third_energy > 0.18f &&
    f.middle_third_energy > 0.22f &&
    f.lower_third_energy > 0.18f;

// ---------------------------------------------------------------------
// Artifact penalty scores — these reduce all class scores for blobs
// that match common non-human heat signatures
// ---------------------------------------------------------------------
  // Small very hot blob — likely a hot component or LED, not a person
  float tinyHotBlobPenalty =
    clamp01(scoreBand(10.0f - f.hot_pixel_count, 0.0f, 6.0f) *
            scoreBand(f.max_target_temp - f.mean_target_temp, 1.5f, 6.0f));

  // Overly filled bounding box — human blobs have internal variation;
  // a nearly solid hot rectangle is more likely a heated surface
  float overCompactPenalty = scoreBand(f.fill_ratio, 0.78f, 0.97f);

  // Wide flat hot bar — heating vents, hot pipes, or shelf edges
  float flatHotBarPenalty =
    clamp01(scoreBand(f.aspect_ratio, 2.0f, 4.0f) *
            scoreBand(4.0f - f.bbox_h, 0.0f, 2.5f));
```

```
  // Single very hot pixel surrounded by cooler pixels — spot heat source
  float singleHotspotPenalty =
    clamp01(scoreBand(f.max_target_temp - f.mean_target_temp, 2.5f, 7.0f) *
            scoreBand(12.0f - f.hot_pixel_count, 0.0f, 8.0f));

// ---------------------------------------------------------------------
// Shared feature scores used across multiple class scorers
// ---------------------------------------------------------------------

  float compact = scoreBand(f.fill_ratio, 0.15f, 0.65f);
  float contrast = scoreBand(f.contrast_above_local_background,
cfg.min_contrast_c, cfg.min_contrast_c + 2.5f);
  float top_heavy = scoreBand(f.top_half_energy, 0.55f, 0.82f);
  float bottom_heavy = scoreBand(f.bottom_half_energy, 0.55f, 0.82f);
  float tall = scoreBand((float)f.bbox_h, 4.0f, 12.0f);
  float wide = scoreBand((float)f.bbox_w, 4.0f, 12.0f);
  float standing_shape = clamp01((cfg.standing_aspect_max - f.aspect_ratio) /
cfg.standing_aspect_max);
  float laying_shape = scoreBand(f.aspect_ratio, cfg.laying_aspect_min, 3.0f);
  float seated_shape = scoreBand(f.aspect_ratio, cfg.seated_aspect_min, 1.25f);
  float crouch_shape = scoreBand(f.aspect_ratio, cfg.crouch_aspect_min, 1.6f);
  float lowerBodyVerticality =
    scoreBand((float)f.bbox_h - (float)f.bbox_w, 1.0f, 6.0f);
  float lowerBodyTaper =
    scoreBand(f.lower_third_width - f.middle_third_width + 1.0f, 0.0f, 3.0f);
  float seatedCentroidDrop =
    scoreBand(f.centroid_y - (f.bbox_y + f.bbox_h * 0.45f), 0.0f, f.bbox_h * 0.35f +
0.1f);
  float seatedLowerWidthBias =
    scoreBand(f.lower_third_width - f.top_third_width + 2.0f, 0.0f, 4.0f);
  float layingWidthDominance =
    scoreBand((float)(f.bbox_w - f.bbox_h), 2.0f, 8.0f);
  float layingEnergyBalance =
    scoreBand(f.middle_third_energy + f.lower_third_energy - f.top_third_energy,
0.0f, 0.50f);

// ---------------------------------------------------------------------
// Per-class scores — each is a weighted dot product of relevant features.
// Weights are empirically tuned; see the tuning guide for adjustment tips.
// ---------------------------------------------------------------------

  s.class_scores[HEAD_ONLY] =
    0.28f * clamp01((8.0f - f.hot_pixel_count) / 6.0f) +
    0.22f * compact +
    0.18f * top_heavy +
```

```
    0.18f * hottest_near_top +
    0.08f * contrast +
    0.06f * meanTempScore;

  if (headOvalShape) {
    s.class_scores[HEAD_ONLY] = clamp01(s.class_scores[HEAD_ONLY] +
0.14f);
  }

  s.class_scores[HEAD_SHOULDERS] =
    0.20f * top_heavy +
    0.22f * scoreBand(f.top_third_width - f.middle_third_width + 2.0f, 0.0f, 4.0f) +
    0.18f * hottest_near_top +
    0.14f * hottest_near_center +
    0.08f * contrast +
    0.12f * compact +
    0.06f * meanTempScore;

  s.class_scores[SHOULDERS_TORSO] =
    0.22f * scoreBand(f.top_third_width - f.lower_third_width + 1.0f, 0.0f, 5.0f) +
    0.18f * scoreBand(f.top_third_energy + f.middle_third_energy -
f.lower_third_energy, 0.0f, 0.5f) +
    0.18f * tall +
    0.20f * compact +
    0.14f * contrast +
    0.06f * meanTempScore;

  s.class_scores[LOWER_BODY] =
    0.24f * bottom_heavy +
    0.18f * scoreBand(f.lower_third_energy - f.top_third_energy + 0.10f, 0.0f,
0.35f) +
    0.14f * scoreBand(f.lower_third_width - f.top_third_width + 2.0f, 0.0f, 4.0f) +
    0.18f * lowerBodyVerticality +
    0.12f * lowerBodyTaper +
    0.08f * contrast +
    0.04f * compact +
    0.02f * meanTempScore;

  s.class_scores[FULL_BODY] =
    0.26f * tall * standing_shape +
    0.18f * scoreBand((float)(f.bbox_h - f.bbox_w), 1.0f, 8.0f) +
    0.16f * scoreBand(0.30f - fabsf(f.top_third_energy - f.lower_third_energy), 0.0f,
0.30f) +
    0.16f * compact +
    0.10f * contrast +
    0.10f * f.temporal_persistence +
```

```
    0.08f * meanTempScore;

  s.class_scores[SITTING] =
    0.24f * seated_shape +
    0.20f * seatedLowerWidthBias +
    0.20f * seatedCentroidDrop +
    0.14f * contrast +
    0.10f * compact +
    0.08f * lowerBodyVerticality +
    0.04f * meanTempScore;


  s.class_scores[CROUCHING] =
    0.26f * crouch_shape +
    0.20f * compact +
    0.18f * scoreBand(f.bbox_h, 3.0f, 8.0f) +
    0.16f * contrast +
    0.12f * scoreBand(1.2f - f.aspect_ratio, 0.0f, 0.5f) +
    0.08f * meanTempScore;

  s.class_scores[LAYING_DOWN] =
    0.28f * laying_shape +
    0.18f * wide +
    0.18f * layingWidthDominance +
    0.14f * layingEnergyBalance +
    0.12f * contrast +
    0.06f * compact +
    0.04f * meanTempScore;

// ---------------------------------------------------------------------
// UNKNOWN_HUMAN_BLOB score — used as a catch-all for human-like blobs
// that don't match any specific body-part shape but pass temperature and
// compactness checks. Lower weight on shape, higher on thermal plausibility.
// ---------------------------------------------------------------------

  float verticalHumanBias =
    scoreBand(f.top_half_energy, 0.50f, 0.72f) *
    scoreBand((float)f.bbox_h, 4.0f, 10.0f);

  float widthBalance =
    1.0f - clamp01(fabsf(f.middle_third_width - f.top_third_width) / max(1.0f,
(float)f.bbox_w));

  float pixelDensityScore = 0.0f;
  if (isfinite(f.distance_estimate_m) && f.distance_estimate_m > 0.2f) {
```

```
    float expectedPixels = 120.0f / (f.distance_estimate_m *
f.distance_estimate_m);
    pixelDensityScore =
      clamp01(1.0f - fabsf((float)f.hot_pixel_count - expectedPixels) / max(1.0f,
expectedPixels));
  } else {
    pixelDensityScore = scoreBand((float)f.hot_pixel_count, 4.0f, 40.0f);
  }
  float clusteredBlobScore =
    clamp01(
      0.34f * compact +
      0.22f * contrast +
      0.18f * temperaturePlausibility +
      0.14f * pixelDensityScore +
      0.12f * clamp01(1.0f - f.size_penalty)
    );

  s.class_scores[UNKNOWN_HUMAN_BLOB] =
    clamp01(
      0.18f * compact +
      0.16f * contrast +
      0.10f * f.temporal_persistence +
      0.08f * f.stability +
      0.14f * verticalHumanBias +
      0.12f * widthBalance +
      0.10f * temperaturePlausibility +
      0.12f * pixelDensityScore +
      0.10f * clusteredBlobScore
    );

// ------------------------------------------------------------------
// Apply artifact penalties to all human class scores
// Each penalty reduces scores proportionally — a strong penalty (e.g., 0.8)
// will reduce a 0.7 score to about 0.42
// ------------------------------------------------------------------

  float nonHumanPenalty =
    0.30f * tinyHotBlobPenalty +
    0.25f * overCompactPenalty +
    0.25f * flatHotBarPenalty +
    0.20f * singleHotspotPenalty;

  for (int i = 0; i < 9; i++) {
    s.class_scores[i] = clamp01(s.class_scores[i] - 0.35f * nonHumanPenalty);
  }
```

```
// ---------------------------------------------------------------------
// Shape confirmation gates — classes with strict shape requirements get
// their scores heavily penalized if the blob doesn't pass the geometric check.
// This prevents a warm blob that's slightly the wrong shape from confidently
// matching a class it couldn't possibly represent.
// ---------------------------------------------------------------------

  bool lowerBodyShape =
    f.bbox_h >= 4 &&
    f.bottom_half_energy >= 0.56f &&
    f.lower_third_width >= max(1.0f, f.top_third_width * 0.90f) &&
    f.aspect_ratio >= 0.35f && f.aspect_ratio <= 1.45f &&
    ((f.bbox_h - f.bbox_w) >= 1) &&
    f.lower_third_energy > f.middle_third_energy * 0.92f;

  if (!lowerBodyShape) {
    s.class_scores[LOWER_BODY] *= 0.25f;
  }

  bool sittingShape =
    f.bbox_h >= 4 &&
    f.bbox_w >= 3 &&
    f.aspect_ratio >= 0.60f && f.aspect_ratio <= 1.20f &&
    seatedCentroidDrop >= 0.18f &&
    seatedLowerWidthBias >= 0.16f &&
    f.bottom_half_energy >= 0.50f;

  bool layingDownShape =
    f.bbox_w >= 4 &&
    f.aspect_ratio >= 1.65f && f.aspect_ratio <= 3.20f &&
    layingWidthDominance >= 0.16f &&
    layingEnergyBalance >= 0.12f &&
    f.bbox_h >= 2 && f.bbox_h <= 8;

  if (!layingDownShape) s.class_scores[LAYING_DOWN] *= 0.22f;
  if (!fullBodyShape) s.class_scores[FULL_BODY] *= 0.38f;
// ---------------------------------------------------------------------
// Scale all class scores by temperature plausibility.
// A blob with perfect shape but wrong temperature still gets reduced scores.
// 0.65 + 0.35 * temp means even a zero-temp-score blob retains 65% of its
// shape score — prevents complete collapse when thermal data is ambiguous.
// ---------------------------------------------------------------------
  for (int i = 0; i < 9; i++) {
    s.class_scores[i] =
      clamp01(s.class_scores[i] * (0.65f + 0.35f * temperaturePlausibility));
  }
```

```
  for (int i = 0; i < 9; i++) {
    s.class_scores[i] = clamp01(s.class_scores[i] - f.size_penalty * 0.18f);
  }

// ---------------------------------------------------------------------
// Find the best specific class (excluding UNKNOWN_HUMAN_BLOB and
NO_HUMAN)
// If a specific class scores above 0.42, reduce the UNKNOWN catch-all so
// the more specific label is preferred in the output
// ---------------------------------------------------------------------
  float best_specific = 0.0f;
  ThermalClassId best_specific_class = HEAD_ONLY;
  for (int i = 0; i < UNKNOWN_HUMAN_BLOB; i++) {
    if (s.class_scores[i] > best_specific) {
      best_specific = s.class_scores[i];
      best_specific_class = (ThermalClassId)i;
    }
  }

  if (best_specific >= 0.42f) {
    s.class_scores[UNKNOWN_HUMAN_BLOB] *= 0.72f;
  }
  if (best_specific > s.class_scores[UNKNOWN_HUMAN_BLOB] + 0.08f) {
    s.class_scores[best_specific_class] =
      clamp01(s.class_scores[best_specific_class] + 0.08f);
  }

// ---------------------------------------------------------------------
// Find the overall winner among all 9 human classes
// ---------------------------------------------------------------------
  float best = 0.0f;
  for (int i = 0; i < 9; i++) {
    if (s.class_scores[i] > best) {
      best = s.class_scores[i];
      s.best_class = (ThermalClassId)i;
    }
  }

// ---------------------------------------------------------------------
// Compute final human_presence_score — drives the EMA in
smoothClassification
// Combines the best class score with shape-agnostic blob quality metrics
// ---------------------------------------------------------------------
  s.human_presence_score = clamp01(
    best * 0.62f +
```

```
    contrast * 0.16f +
    compact * 0.10f +
    f.temporal_persistence * 0.06f +
    f.stability * 0.06f
  );

  // Temperature failure: if the blob is not in the human thermal band,
  // heavily suppress the score and force NO_HUMAN output
  if (!tempBandOk) {
    s.human_presence_score *= 0.30f;
    s.best_class = NO_HUMAN;
  } else if (!(headOvalShape || headShouldersShape || torsoLegsShape ||
fullBodyShape)) {
    float clusteredShapeFallback =
      clamp01(
        0.40f * compact +
        0.24f * contrast +
        0.18f * temperaturePlausibility +
        0.18f * clamp01(1.0f - f.size_penalty)
      );
    s.human_presence_score *= (0.55f + 0.35f * clusteredShapeFallback);
  }
  // NO_HUMAN score is the complement of human_presence_score
  s.class_scores[NO_HUMAN] = clamp01(1.0f - s.human_presence_score);
  // Final confidence: presence score weighted by stability and persistence
  s.confidence = clamp01(
    s.human_presence_score * 0.70f +
    f.stability * 0.15f +
    f.temporal_persistence * 0.15f
  );
}

// ---------------------------------------------------------------------
// smoothClassification — apply EMA temporal smoothing to classification output
//
// Updates the presence EMA and per-class EMAs in ThermalState using
// asymmetric rise/fall alphas so detection builds quickly but decays slowly.
// Also tracks centroid stability across frames — stable_frames increments
// when the centroid moves < 4 pixels, decrements otherwise.
//
// After smoothing, overwrites the ThermalScores output with the EMA values
// so the caller sees temporally smoothed scores rather than per-frame noise.
// ---------------------------------------------------------------------
static inline void smoothClassification(ThermalState& state, const
ThermalConfig& cfg, ThermalFeatures& f, ThermalScores& s) {
  // Use rise_alpha when presence is increasing, fall_alpha when decreasing
```

```
  float alpha = (s.human_presence_score >= state.presence_ema) ?
cfg.rise_alpha : cfg.fall_alpha;
  state.presence_ema += alpha * (s.human_presence_score -
state.presence_ema);

  // Apply same asymmetric alpha to each class score EMA
  for (int i = 0; i < kClassCount; i++) {
    state.class_ema[i] += alpha * (s.class_scores[i] - state.class_ema[i]);
  }

  // Update centroid stability counter
  float jump = sqrtf(
    (f.centroid_x - state.prev_centroid_x) * (f.centroid_x - state.prev_centroid_x) +
    (f.centroid_y - state.prev_centroid_y) * (f.centroid_y - state.prev_centroid_y)
  );
  if (jump < 4.0f) {
    if (state.stable_frames < 30) state.stable_frames++;
  } else if (state.stable_frames > 0) {
    state.stable_frames--;
  }
  state.prev_centroid_x = f.centroid_x;
  state.prev_centroid_y = f.centroid_y;

  // Overwrite output scores with smoothed EMA values
  s.best_class = NO_HUMAN;
  float best = 0.0f;
  for (int i = 0; i < 9; i++) {
    s.class_scores[i] = state.class_ema[i];
    if (s.class_scores[i] > best) {
      best = s.class_scores[i];
      s.best_class = (ThermalClassId)i;
    }
  }
  s.human_presence_score = state.presence_ema;
  s.class_scores[NO_HUMAN] = clamp01(1.0f - s.human_presence_score);

  // Final confidence: presence score + stability bonus + class specificity bonus
  // Specific named classes (not UNKNOWN) get a higher confidence bonus
  s.confidence = clamp01(
    s.human_presence_score * 0.55f +
    clamp01(state.stable_frames / 12.0f) * 0.25f +
    (s.best_class != NO_HUMAN && s.best_class !=
UNKNOWN_HUMAN_BLOB ? 0.20f : 0.08f)
);
  // Update temporal features in the feature struct from smoothed state
  f.temporal_persistence = clamp01(state.stable_frames / 10.0f);
```

```
  f.confidence = s.confidence;
}

// ---------------------------------------------------------------------
// detectHumanThermal — top-level entry point
//
// Runs the full detection pipeline for one frame:
//   1. Spatial smooth
//   2. Scene statistics
//   3. Indoor/outdoor mode selection
//   4. Background update
//   5. Blob segmentation
//   6. Feature extraction + classification for each blob
//   7. Select best blob by presence score
//   8. Temporal smoothing
//   9. Final hysteresis threshold decision
//
// Parameters:
//   state            — persistent state (must be initialized with initState())
//   frame_c          — raw 768-pixel temperature array in °C from MLX90640
//   timestamp_ms     — millis() timestamp (reserved for future use)
//   sensor_ambient_c  — ambient temperature reported by MLX90640 sensor header
//   estimated_distance_m — subject distance from external ranging, or NAN
//   cfg_in           — configuration preset (use hybridDefaults() if unsure)
//   best_features    — filled with extracted features of the best blob on return
//   best_scores      — filled with classification scores of the best blob on return
//
// Returns: true if a human is detected with sufficient confidence and presence score
// ---------------------------------------------------------------------
inline bool detectHumanThermal(
  ThermalState& state,
  const float* frame_c,
  uint32_t timestamp_ms,
  float sensor_ambient_c,
  float estimated_distance_m,
  const ThermalConfig& cfg_in,
  ThermalFeatures& best_features,
  ThermalScores& best_scores
) {
  ThermalConfig cfg = cfg_in;
  smoothFrame3x3(frame_c, state.smoothed);  // Step 1: spatial smooth to reduce pixel-level thermal noise
```

```
  float min_t, max_t, mean_t, p75, p90;     // Step 2: compute scene statistics for adaptive thresholding
  computeFrameStats(state.smoothed, min_t, max_t, mean_t, p75, p90);

  // Step 3: auto-detect outdoor conditions from scene temperature statistics.
  // Outdoor mode uses higher contrast thresholds to compensate for warm surfaces.
  // Once outdoor mode activates it uses a stickier hysteresis to avoid rapid toggling.
  if (cfg.force_outdoor) {
    state.auto_outdoor = true;
  } else {
    const bool enterOutdoor =
      sensor_ambient_c > 28.0f || mean_t > 30.0f || (p90 - p75) > 1.8f;
    const bool stayOutdoor =
      sensor_ambient_c > 27.0f || mean_t > 29.0f || (p90 - p75) > 1.3f;
      // Hysteresis: harder to enter outdoor mode than to stay in it
    state.auto_outdoor = state.auto_outdoor ? stayOutdoor : enterOutdoor;
  }
  if (state.auto_outdoor) cfg = outdoorDefaults();

  // Step 4: update per-pixel background model
  updateBackground(state, state.smoothed, timestamp_ms, sensor_ambient_c, cfg);

  // Step 5: threshold and flood-fill to find candidate blobs
  int blob_count = segmentBlobs(state, state.smoothed, p75, p90, sensor_ambient_c, estimated_distance_m, cfg);

  // Initialize outputs to no-detection state
  memset(&best_features, 0, sizeof(best_features));
  memset(&best_scores, 0, sizeof(best_scores));
  best_scores.best_class = NO_HUMAN;
  float best_metric = 0.0f;

  // Steps 6-7: process each valid blob and keep the one with the highest presence score
  for (int label = 0; label < blob_count; label++) {
    ThermalFeatures features{};
    ThermalScores scores{};
    extractFeatures(state, state.smoothed, label, p75, estimated_distance_m, cfg, features);
    if (features.hot_pixel_count <= 0) continue;

    // Pre-filter blobs that violate fill ratio or edge compactness bounds
```

```
    if (features.fill_ratio < cfg.fill_ratio_min || features.fill_ratio > cfg.fill_ratio_max)
continue;
    if (features.edge_compactness < cfg.edge_compactness_min) continue;

    classifyThermalBlob(features, cfg, scores);

    // Selection metric: presence score weighted by normalized area
    float metric = scores.human_presence_score * (0.70f + min(1.0f,
features.hot_pixel_count / 24.0f) * 0.30f);
    if (metric > best_metric) {
      best_metric = metric;
      best_features = features;
      best_scores = scores;
    }
  }

  // No valid blobs found — decay presence EMA and return no-detection
  if (best_metric <= 0.0f) {
    state.presence_ema = max(0.0f, state.presence_ema - cfg.fall_alpha);
    for (int i = 0; i < kClassCount; i++) state.class_ema[i] *= 0.92f;
    memset(&best_features, 0, sizeof(best_features));
    memset(&best_scores, 0, sizeof(best_scores));
    best_scores.human_presence_score = state.presence_ema;
    best_scores.class_scores[NO_HUMAN] = 1.0f;
    best_scores.best_class = NO_HUMAN;
    best_scores.confidence = 1.0f - best_scores.human_presence_score;
    snprintf(best_features.debug, sizeof(best_features.debug), "no_blob p75=%.2f
p90=%.2f amb=%.2f", p75, p90, sensor_ambient_c);
    return best_scores.human_presence_score >= cfg.present_on;
  }
  // Step 8: temporal smoothing — updates EMA and overwrites best_scores with
smoothed values
  smoothClassification(state, cfg, best_features, best_scores);

  // Step 9: final hysteresis decision
  bool generic_blob_only = (best_scores.best_class ==
UNKNOWN_HUMAN_BLOB);
  bool shapeConsensus =
    best_scores.best_class == HEAD_ONLY ||
    best_scores.best_class == HEAD_SHOULDERS ||
    best_scores.best_class == SHOULDERS_TORSO ||
    best_scores.best_class == LOWER_BODY ||
    best_scores.best_class == FULL_BODY ||
    best_scores.best_class == SITTING ||
    best_scores.best_class == LAYING_DOWN;
  const float humanTempVariationC = cfg.human_temp_variation_f * (5.0f / 9.0f);
```

```
  // Recompute temperature scores on the best blob for the final gate
  float meanFloorScore = scoreBand(best_features.mean_target_temp,
cfg.min_mean_human_temp_c - 1.5f, cfg.min_mean_human_temp_c + 2.0f);
  float meanBandScore =
    1.0f - clamp01(fabsf(best_features.mean_target_temp -
cfg.human_temp_center_c) / max(0.5f, humanTempVariationC + 1.5f));
  float maxBandScore =
    1.0f - clamp01(fabsf(best_features.max_target_temp -
cfg.human_temp_center_c) / max(0.5f, humanTempVariationC + 3.0f));

  float hotspotPenalty = 0.0f;
  if (isfinite(best_features.max_target_temp) &&
best_features.max_target_temp > cfg.max_hotspot_temp_c) {
    hotspotPenalty = clamp01((best_features.max_target_temp -
cfg.max_hotspot_temp_c) / 8.0f);
  }
  float finalTempScore =
  clamp01(0.45f * meanFloorScore + 0.35f * meanBandScore + 0.20f *
maxBandScore - 0.35f * hotspotPenalty);

  // Cold ambient shape mode
  const float coldAmbientOnlyShapeC = (74.0f - 32.0f) * (5.0f / 9.0f);
  bool coldAmbientShapeMode =
    isfinite(sensor_ambient_c) &&
    sensor_ambient_c < coldAmbientOnlyShapeC;

  // Soft shape consensus
  bool softShapeConsensus =
    shapeConsensus ||
    (best_scores.best_class == UNKNOWN_HUMAN_BLOB &&
     best_features.fill_ratio >= 0.18f &&
     best_features.hot_pixel_count >= 4 &&
     best_features.aspect_ratio >= 0.45f &&
     best_features.aspect_ratio <= 2.1f &&
     best_features.contrast_above_local_background >= cfg.min_contrast_c);

  // Classify which body-part group this is for threshold adjustment
  bool partialBodyClass =
    best_scores.best_class == HEAD_SHOULDERS ||
    best_scores.best_class == SHOULDERS_TORSO ||
    best_scores.best_class == LOWER_BODY;
  bool headFocusedClass =
    best_scores.best_class == HEAD_ONLY ||
    best_scores.best_class == HEAD_SHOULDERS;
```

```
// Adaptive thresholds
float on_threshold = cfg.present_on + (generic_blob_only ? 0.04f : 0.0f) -
(partialBodyClass ? 0.10f : 0.0f) - (headFocusedClass ? 0.14f : 0.0f);
float off_threshold = cfg.present_off + (generic_blob_only ? 0.03f : 0.0f) -
(partialBodyClass ? 0.08f : 0.0f) - (headFocusedClass ? 0.10f : 0.0f);
float min_confidence = (generic_blob_only ? 0.64f : 0.56f) - (partialBodyClass ?
0.10f : 0.0f) - (headFocusedClass ? 0.12f : 0.0f);
 // Enforce absolute floor on thresholds to prevent degenerate configs
on_threshold = max(0.38f, on_threshold);
off_threshold = max(0.28f, off_threshold);
min_confidence = max(0.34f, min_confidence);

bool shapeOk = softShapeConsensus;
  bool strongShapeClass =
    best_scores.best_class == HEAD_ONLY ||
    best_scores.best_class == HEAD_SHOULDERS ||
    best_scores.best_class == SHOULDERS_TORSO ||
    best_scores.best_class == LOWER_BODY ||
    best_scores.best_class == FULL_BODY ||
    best_scores.best_class == SITTING ||
    best_scores.best_class == LAYING_DOWN;

// Temperature gate — passes if:
//   a) In cold ambient mode with a strong specific shape class (shape overrides
temp), OR
//   b) finalTempScore meets the class-appropriate minimum threshold
bool tempOk =
  (coldAmbientShapeMode && strongShapeClass && shapeConsensus) ||
  (finalTempScore >= (generic_blob_only ? 0.42f : (headFocusedClass ? 0.28f :
(partialBodyClass ? 0.30f : 0.38f))));

// Outdoor shape assist: in outdoor mode, a high-confidence specific shape class
// can bypass the normal temperature gate since outdoor thermal signatures are
// more variable due to wind, clothing, and sun-heated surfaces
bool outdoorShapeAssist =
  state.auto_outdoor &&
  strongShapeClass &&
  shapeConsensus &&
  (best_scores.confidence >= 0.55f ||
   (finalTempScore >= 0.14f &&
    best_scores.human_presence_score >= max(0.12f, on_threshold - 0.24f) &&
    best_scores.confidence >= max(0.22f, min_confidence - 0.16f)));

// Final detection decision — true if:
//   Shape passes AND (temperature passes OR outdoor shape assist)
//   AND (presence score above on_threshold with sufficient confidence
```

```
//        OR outdoor shape assist bypasses
//        OR hysteresis holdover keeps detection active above off_threshold)
return shapeOk && (tempOk || outdoorShapeAssist) &&
       (((best_scores.human_presence_score >= on_threshold) &&
(best_scores.confidence >= min_confidence)) ||
        outdoorShapeAssist ||
        ((state.presence_ema >= off_threshold) && (best_scores.confidence >=
(min_confidence - 0.08f))));
}

// ---------------------------------------------------------------------
// printDebug — formatted single-line debug output for Serial or log
// Prints all key detection metrics in a compact human-readable format
// ---------------------------------------------------------------------
static inline void printDebug(Stream& stream, uint32_t ts, const
ThermalFeatures& f, const ThermalScores& s) {
  stream.printf(
    "THERM ts=%lu present=%.2f class=%s conf=%.2f bbox=%d,%d,%d,%d
ctr=%.1f,%.1f area=%d asp=%.2f fill=%.2f max=%.2f mean=%.2f bg=%.2f
dT=%.2f pers=%.2f stab=%.2f\n",
    (unsigned long)ts, s.human_presence_score, className(s.best_class),
s.confidence,
    f.bbox_x, f.bbox_y, f.bbox_w, f.bbox_h, f.centroid_x, f.centroid_y,
f.hot_pixel_count,
    f.aspect_ratio, f.fill_ratio, f.max_target_temp, f.mean_target_temp,
f.local_background,
    f.contrast_above_local_background, f.temporal_persistence, f.stability
  );
}

/*
============================================================
==============
TUNING GUIDE
============================================================
==============
Work through parameters in this order — each layer depends on the one above.

1. SEGMENTATION (bg_delta_c, percentile_delta_c, min_contrast_c)
   Goal: clean binary mask with human pixels hot, background cold.
   Too low → whole warm wall shows as a blob.
   Too high → human pixels drop out at distance or when wearing heavy clothing.

2. BLOB REJECTION (min_blob_area_near, min_blob_area_far,
edge_compactness_min,
                   fill_ratio_min, fill_ratio_max)
```

Goal: reject heaters, pipes, and random thermal noise before classification.
Log the debug string for false-positive blobs and look at their fill_ratio
and edge_compactness to find the right cutoff.

3. CLASSIFICATION SHAPES (laying_aspect_min, seated_aspect_min,
crouch_aspect_min, standing_aspect_max)
Goal: correct class label on your most common use cases.
Print className(scores.best_class) while walking through each pose.

4. TEMPERATURE PARAMETERS (human_temp_center_c,
human_temp_variation_f,
max_hotspot_temp_c, min_mean_human_temp_c)
Goal: pass clothed humans at your deployment temperature range.
In cold weather (< 10°C ambient) lower human_temp_center_c to ~30°C.
In hot weather (> 30°C ambient) increase min_contrast_c.

5. HYSTERESIS (present_on, present_off, rise_alpha, fall_alpha)
Goal: stable on/off transitions without chattering.
If detection fires then drops immediately → increase rise_alpha.
If detection holds too long after person leaves → increase fall_alpha.
If too many false positives → increase present_on.
If real detections are missed → decrease present_on.

==========
*/

}

#endif

7. Initiator UWB.cpp header

```
#define MAIN_U1
#include "uwb.h"
#include "dw3000.h"

static const uint32_t UWB_WAIT_BUDGET_US = 50000;

// Config matches the known-good role layout and timing path.
dwt_config_t config = {
    5,
    DWT_PLEN_128,
    DWT_PAC8,
```

```
    9,
    9,
    1,
    DWT_BR_6M8,
    DWT_PHRMODE_STD,
    DWT_PHRRATE_STD,
    (129 + 8 - 8),
    DWT_STS_MODE_OFF,
    DWT_STS_LEN_64,
    DWT_PDOA_M0
};

extern dwt_txconfig_t txconfig_options;

uint8_t tx_msg[] = {0x41, 0x88, 0, 0xCA, 0xDE, 0, 0, UID, 0, FUNC_CODE_INTER,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0};

uint8_t rx_msg[] = {0x41, 0x88, 0, 0xCA, 0xDE, 0, 0, UID, 0, FUNC_CODE_INTER,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0};

uint8_t frame_seq_nb = 0;
uint8_t rx_buffer[NUM_NODES - 1][BUF_LEN];
uint32_t status_reg = 0;
int counter = 0;
int ret;
uint64_t rx_ts[NUM_NODES - 1];
uint64_t desired_tx_ts, real_tx_ts;
uint64_t tx_time;
uint64_t t_round;
double tof, distance;
unsigned long previous_debug_millis = 0;
unsigned long current_debug_millis = 0;
int millis_since_last_serial_print;
```

```
int target_uids[NUM_NODES - 1];
const char* uwb_last_error = "UWB:idle";

static void set_target_uids() {
#ifdef MAIN_U1
    target_uids[0] = U2;
#elif defined(MAIN_U2)
    target_uids[0] = U1;
#endif
}

bool start_uwb() {
    // setupUWB() already waited for IDLE_RC before calling here — no repeat needed
    uwb_last_error = "UWB:idle";

    if (dwt_initialise(DWT_DW_INIT) == DWT_ERROR) {
        if (dwt_initialise(DWT_DW_IDLE_RC) == DWT_ERROR) {
            uwb_last_error = "UWB:init hw";
            return false;
        }
    }
    dwt_setxtaltrim(0x2E);
    dwt_setleds(DWT_LEDS_DISABLE);

    if (dwt_configure(&config)) {
        uwb_last_error = "UWB:cfg";
        return false;
    }

    dwt_configuretxrf(&txconfig_options);
    dwt_setrxantennadelay(RX_ANT_DLY);
    dwt_settxantennadelay(TX_ANT_DLY);
    dwt_setrxaftertxdelay(TX_TO_RX_DLY_UUS);
#ifdef INITIATOR
    dwt_setrxtimeout(RX_TIMEOUT_UUS);
#else
    dwt_setrxtimeout(0);
```

```
#endif

    dwt_setlnapamode(DWT_LNA_ENABLE | DWT_PA_ENABLE);
    set_target_uids();
    uwb_last_error = "UWB:ready";
    return true;
}

void initiator() {
    const uint32_t waitStartUs = micros();
    if (counter == 0) {
        tx_msg[MSG_SN_IDX] = frame_seq_nb;
        tx_msg[MSG_FUNC_IDX] = FUNC_CODE_INTER;
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_TXFRS_BIT_MASK);
        dwt_writetxdata((uint16_t)(MSG_LEN), tx_msg, 0);
        dwt_writetxfctrl((uint16_t)(MSG_LEN), 0, 1);
        int txret = dwt_starttx(DWT_START_TX_IMMEDIATE | DWT_RESPONSE_EXPECTED);
        if (txret != DWT_SUCCESS) {
            uwb_last_error = "UWB:poll tx";
            counter = 0;
            ++frame_seq_nb;
            return;
        }
    } else {
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_TXFRS_BIT_MASK);
        dwt_rxenable(DWT_START_RX_IMMEDIATE);
    }

    while (!((status_reg = dwt_read32bitreg(SYS_STATUS_ID)) &
             (SYS_STATUS_RXFCG_BIT_MASK | SYS_STATUS_ALL_RX_TO |
              SYS_STATUS_ALL_RX_ERR))) {
        if ((micros() - waitStartUs) > UWB_WAIT_BUDGET_US) {
            status_reg = SYS_STATUS_ALL_RX_TO;
            break;
        }
    }
    if (status_reg & SYS_STATUS_RXFCG_BIT_MASK) {
```

```
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_RXFCG_BIT_MASK);
        dwt_readrxdata(rx_buffer[counter], BUF_LEN, 0);
        rx_ts[counter] = get_rx_timestamp_u64();
        if (rx_buffer[counter][MSG_SID_IDX] != target_uids[counter]) {
          Serial.print("RX SID=");
Serial.print(rx_buffer[counter][MSG_SID_IDX]);
Serial.print(" expected=");
Serial.println(target_uids[counter]);

          uwb_last_error = "UWB:src mismatch";
          dwt_write32bitreg(SYS_STATUS_ID,
                    SYS_STATUS_ALL_RX_TO | SYS_STATUS_ALL_RX_ERR);
          counter = 0;
          ++frame_seq_nb;
          return;
        }
        ++counter;
      } else {
        uwb_last_error = "UWB:rx timeout";
        tx_msg[MSG_SN_IDX] = frame_seq_nb;
        tx_msg[MSG_FUNC_IDX] = FUNC_CODE_RESET;
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_TXFRS_BIT_MASK);
        dwt_writetxdata((uint16_t)(MSG_LEN), tx_msg, 0);
        dwt_writetxfctrl((uint16_t)(MSG_LEN), 0, 1);
        dwt_starttx(DWT_START_TX_IMMEDIATE);
        dwt_write32bitreg(SYS_STATUS_ID,
                  SYS_STATUS_ALL_RX_TO | SYS_STATUS_ALL_RX_ERR);
        counter = 0;
        ++frame_seq_nb;
        delay(1);
        return;
      }

      if (counter == NUM_NODES - 1) {
        float clockOffsetRatio;
        clockOffsetRatio = ((float)dwt_readclockoffset()) / (uint32_t)(1 << 26);
        real_tx_ts = get_tx_timestamp_u64();
```

```
        for (int i = 0; i < counter; i++) {
            uint32_t t_reply_raw = 0;
            resp_msg_get_ts(&rx_buffer[i][UID], &t_reply_raw);
            uint64_t t_reply = (uint64_t)t_reply_raw;
            t_round = rx_ts[i] - real_tx_ts;
            if ((int64_t)t_round < 0) t_round += (1ULL << 40);
            tof = ((t_round - t_reply * (1 - clockOffsetRatio)) / 2.0) *
                  DWT_TIME_UNITS;
            distance = tof * SPEED_OF_LIGHT;
        }
        uwb_last_error = "UWB:ready";
        previous_debug_millis = current_debug_millis;
        counter = 0;
        ++frame_seq_nb;
    }
}
```

8. UWB header

```
/*
 * UWB ranging implementation for DW3000
 * Based on the DW3000 Arduino library by NConcepts
 * Repository: https://github.com/Makerfabs/Makerfabs-ESP32-UWB-DW3000
 * Maintained by Makerfabs
 * Adapted for ESP32 multi-sensor SAR device with shared SPI bus,
 * CAT9555 GPIO expander reset, and FreeRTOS mutex arbitration.
 */

#pragma once
#ifndef UWB_H
#define UWB_H

#include "dw3000.h"
extern SemaphoreHandle_t spiBusMutex;

// ----------------------------
// Global constants
// ----------------------------
```

```
#define NUM_NODES       2
#define INTERVAL        100     // ms – safe value

// Function codes (your original values)
#define FUNC_CODE_INTER 0xE2
#define FUNC_CODE_RESET 0xE3

// Node UIDs
#define U0 0
#define U1 10
#define U2 14

#define UWB_IRQ   34
#define UWB_SS    13
// RST handled via CAT9555 expander in main.cpp

// Role-specific defines
#ifdef MAIN_U1
  #define APP_NAME    "UWB DW3000 U1"
  #define UID         U1
  #define TX_ANT_DLY  16385
  #define RX_ANT_DLY  16385
  #define POLL_NUM    0
  #define INITIATOR
#elif defined(MAIN_U2)
  #define APP_NAME    "UWB DW3000 U2"
  #define UID         U2
  #define TX_ANT_DLY  16385
  #define RX_ANT_DLY  16385
  #define POLL_NUM    1
#else
  #define APP_NAME    "UWB DW3000 UNKNOWN"
  #define UID         0
  #define POLL_NUM    0
#endif

#define REPORT_DISTANCE_FROM POLL_NUM
```

```
// Message structure
#define MSG_LEN            36
#define BUF_LEN            MSG_LEN
#define MSG_SN_IDX         2
#define MSG_SID_IDX        7
#define MSG_FUNC_IDX       9
#define RESP_MSG_TS_LEN    4

// Timings
#define TX_TO_RX_DLY_UUS   100
#define RX_TO_TX_DLY_UUS   1000
#define RX_TIMEOUT_UUS     400000

// ----------------------------
// Extern declarations
// ----------------------------
extern dwt_config_t config;
extern uint8_t tx_msg[], rx_msg[];
extern uint8_t frame_seq_nb;
extern uint8_t rx_buffer[NUM_NODES - 1][BUF_LEN];
extern uint32_t status_reg;
extern int counter;
extern int ret;
extern uint64_t rx_ts[NUM_NODES - 1];
extern uint64_t desired_tx_ts, real_tx_ts;
extern uint64_t tx_time;
extern uint64_t t_reply;
extern uint64_t t_round;
extern double tof, distance;
extern unsigned long previous_debug_millis, current_debug_millis;
extern int target_uids[NUM_NODES - 1];
extern const char* uwb_last_error;

// ----------------------------
// Function prototypes
// ----------------------------
```

```
bool start_uwb();
void initiator();

#endif // UWB_H
```

9. Anchor main code

```
#define MAIN_U2
#define RADAR2_EMBEDDED   // prevents Radar2.ino from defining its own setup/loop
#include <Wire.h>
#include <ArduinoBLE.h>
#include "Radar2Types.h"
#include "Radar2VitalsMsg.h"
#include "uwb.h"
#include "dw3000.h"

// ── BLE service / characteristic ──────────────────────────────────────────
// UUIDs must match Radar2BLEClient.h on the main ESP32 side
#define RADAR2_SERVICE_UUID        "12345678-1234-1234-1234-123456789abc"
#define RADAR2_CHARACTERISTIC_UUID "abcdefab-cdef-abcd-efab-cdefabcdefab"

// 6-byte payload: [flags, bpm, hr, distLo, distMid, distHi]
BLEService        radar2Service(RADAR2_SERVICE_UUID);
BLECharacteristic radar2Char(RADAR2_CHARACTERISTIC_UUID, BLERead | BLENotify, 6);

// ── Radar2 vitals externs (defined in Radar2.ino) ─────────────────────────
extern int          lastBpm;
extern bool         bpmValid;
extern int          lastHrBpm;
extern bool         hrValid;
extern float        filtDistM;
extern bool         hasDistance;
extern bool         motionLockout;
extern unsigned long lastHrGoodMs;

// If Radar2.ino exposes this, it helps re-sync radar when switching back.
// If not available in your build, remove the call in switchToMode().
void radar2RecoverLink();
```

```
// ── Time-sliced mode control ──────────────────────────────────────────────
enum RunMode {
  MODE_UWB,
  MODE_RADAR
};

RunMode runMode = MODE_UWB;
unsigned long modeStartMs = 0;

// Change these to 5UL * 60UL * 1000UL for 5-minute windows
const unsigned long UWB_WINDOW_MS   = 60UL * 1000UL;   // 1 minute
const unsigned long RADAR_WINDOW_MS = 60UL * 1000UL;   // 1 minute

static void switchToMode(RunMode newMode) {
  runMode = newMode;
  modeStartMs = millis();

  if (runMode == MODE_UWB) {
    Serial.println("Switching to UWB mode");
  } else {
    Serial.println("Switching to RADAR mode");
    radar2RecoverLink();
  }
}

// ── Pack and notify vitals over BLE (called from loop) ──────────────────
void packRadar2Vitals() {
  BLE.poll();  // service ArduinoBLE stack

  if (!BLE.connected()) return;

  static unsigned long lastPackMs = 0;
  unsigned long now = millis();
  if (now - lastPackMs < 200) return;  // 5 Hz max
  lastPackMs = now;
```

```
    uint8_t buf[6] = {0};
    uint8_t flags  = 0;

    if (bpmValid && lastBpm > 0) {
        flags |= RADAR2_FLAGS_BPM;
        buf[RADAR2_BPM_IDX] = (uint8_t)constrain(lastBpm, 0, 255);
    }

    bool hrNowValid = hrValid && !motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS);
    if (hrNowValid && lastHrBpm > 0) {
        flags |= RADAR2_FLAGS_HR;
        buf[RADAR2_HR_IDX] = (uint8_t)constrain(lastHrBpm, 0, 255);
    }

    if (hasDistance && isfinite(filtDistM) && filtDistM >= 0.0f) {
        flags |= RADAR2_FLAGS_DIST;
        uint32_t cm = (uint32_t)(filtDistM * 100.0f);
        buf[RADAR2_DIST_IDX]     =  cm        & 0xFF;
        buf[RADAR2_DIST_IDX + 1] = (cm >>  8) & 0xFF;
        buf[RADAR2_DIST_IDX + 2] = (cm >> 16) & 0xFF;
    }

    buf[RADAR2_FLAGS_IDX] = flags;
    radar2Char.writeValue(buf, 6);
}

// ── Arduino entry points ──────────────────────────────────────────────────────
void setup() {
    Serial.begin(115200);
    delay(200);

    Wire.begin();   // I2C for LCD
    spiBegin(UWB_IRQ, UWB_RST);
    spiSelect(UWB_SS);
    delay(2);
```

```
    uint32_t id = dwt_readdevid();
    Serial.print("DW ID: 0x");
    Serial.println(id, HEX);
    start_uwb();

    radar2Setup(false, true);  // Serial already started above

    // BLE server
    if (!BLE.begin()) {
        Serial.println("BLE init failed — continuing without BLE");
    } else {
        BLE.setLocalName("Radar2Anchor");
        BLE.setAdvertisedService(radar2Service);
        radar2Service.addCharacteristic(radar2Char);
        BLE.addService(radar2Service);
        BLE.advertise();
        Serial.println("BLE advertising as Radar2Anchor");
    }

    modeStartMs = millis();
    Serial.println("Starting in UWB mode");
}

void loop() {
    unsigned long now = millis();

    if (runMode == MODE_UWB) {
        responder();

        if (now - modeStartMs >= UWB_WINDOW_MS) {
            switchToMode(MODE_RADAR);
        }

        return;
    }
```

```
    radar2LoopStep();
    packRadar2Vitals();

    if (now - modeStartMs >= RADAR_WINDOW_MS) {
        switchToMode(MODE_UWB);
    }
}
```

10. Waveshare Radar2 code

```
#include "Radar2Types.h"
#include <Arduino.h>
#include <arduinoFFT.h>
#include <math.h>
#include <string.h>
#include "RadarLcd1602.h"

#if defined(ARDUINO_UNOR4_WIFI)
  #define RADAR_RX_PIN 0
  #define RADAR_TX_PIN 1
  #define RADAR_SERIAL_PORT Serial1
#else
  #define RADAR_RX_PIN 16
  #define RADAR_TX_PIN 17
  HardwareSerial RadarSerial(2);
  #define RADAR_SERIAL_PORT RadarSerial
#endif

// ====================== CONFIG ======================
#define SAMPLES              128
#define SAMPLING_FREQ        10.0
#define SAMPLE_PERIOD_MS     100UL
#define FAST_SAMPLES         64     // 6.4 s — enough to resolve 0.15 Hz adult
#define FAST_ANALYZE_PERIOD_MS 500UL  // run fast path every 500 ms
#define FAST_SNR_MIN         2.20f  // stricter — partial window is noisier
#define CARDIAC_SAMPLES      256
#define CARDIAC_FREQ         10.0
#define ANALYZE_PERIOD_MS    250UL
```

```
#define BPM_HOLD_MS          3500UL

// Distance
#define DETECT_MIN_DIST_M    0.10f
#define DETECT_MAX_DIST_M    8.00f
#define BPM_MIN_DIST_M       0.35f
#define BPM_MAX_DIST_M       4.00f
#define DIST_OUTLIER_STEP_M  0.60f
#define DIST_ALPHA           0.06f
#define DETECT_HOLD_MS       5000UL

// Signal / motion
#define DESPIKE_DELTA        1.20f
#define MOTION_VAR_THRESHOLD 0.25

// Mode watchdog
#define MODE_WATCHDOG_MS     3000UL
#define MODE_RETRY_GAP_MS    3000UL

// Anti-drift publish behavior
#define UP_CONFIRM_N         5
#define DOWN_CONFIRM_N       6
#define MAX_UP_STEP_BPM      1
#define MAX_DOWN_STEP_BPM    1

// Classifier behavior (overarching band selector)
#define CLASS_HIST_N         15
#define CLASS_MIN_SAMPLES    2
#define CLASS_SWITCH_VOTES   5
#define FAST_LOCK_SNR        2.60f
#define FAST_ADULT_SNR       2.10f
#define QUIET_SETTLE_MS      300UL
#define ENERGY_SPIKE_LOG_DELTA 1.50
#define DIST_STABLE_HISTORY_N 5
#define DIST_STABLE_SPAN_M   0.30f
#define DIST_ALPHA_BETA_ALPHA 0.55f
#define DIST_ALPHA_BETA_BETA  0.08f
```

```
#define BPM_KALMAN_Q          0.8f
#define BPM_KALMAN_R_BASE     5.0f
#define BAND_LOCK_HOLD_MS     8000UL
#define ADULT_STAY_SNR        1.50f
#define GATE_KEEP_RATIO       0.45f

// Cardiac detection
#define CARDIAC_LOW_HZ        0.83f   // 50 BPM
#define CARDIAC_HIGH_HZ       2.50f   // 150 BPM
#define CARDIAC_SNR_MIN       1.20f   // lower than breathing — signal is weaker
#define CARDIAC_PEAK_MIN      0.05f   // much lower than breathing threshold
#define CARDIAC_NOTCH_WIDTH   2       // bins to null on each side of breathing harmonic
#define CARDIAC_ANALYZE_MS    2000UL  // run cardiac analysis every 2 seconds
#ifndef CARDIAC_HOLD_MS
#define CARDIAC_HOLD_MS       5000UL  // hold last valid HR reading this long
#endif
#define HR_KALMAN_Q           4.0f
#define HR_KALMAN_R           8.0f
#define HR_MEDIAN_N           3
#define HR_MAX_STEP_BPM       6
#define HR_MAX_DOWN_STEP_BPM  2
#define HR_FAST_STEP_SNR      6.0
#define HR_OUTLIER_STEP_BPM   18
#define HR_DROP_GUARD_BPM     12
#define HR_DROP_AC_RATIO      1.35
#define HR_FFT_AGREE_BPM      12
#define HR_ALIAS_AC_RATIO     0.90
#define HR_BAND_SWITCH_VOTES  4
#define HR_BOOTSTRAP_N        3
#define HR_BOOTSTRAP_SPAN_BPM 10
#define HR_BOOTSTRAP_SNR_MIN  2.0
#define CARDIAC_MIN_BPM       65      // min HR for current adult tuning
#define CARDIAC_MAX_BPM       100     // Max HR
#define CARDIAC_REQUIRE_BREATH_LOCK true  // only run when breathing is confident


// Report-mode command (Waveshare HMMD frame mode)
```

```
static const uint8_t CMD_SET_REPORT_MODE[] = {
  0xFD, 0xFC, 0xFB, 0xFA,
  0x08, 0x00,
  0x12, 0x00,
  0x00, 0x00,
  0x04, 0x00, 0x00, 0x00,
  0x04, 0x03, 0x02, 0x01
};

// ====================== BAND CONFIG ======================
static const BandCfg BAND_CFGS[] = {
  { "Adult",  0.15f, 0.38f, 11, 23, 20, 1.55f, 2.05f, 0.75f },
  { "Child",  0.28f, 0.62f, 17, 38, 30, 1.35f, 1.80f, 0.60f },
  { "Infant", 0.50f, 0.95f, 28, 60, 50, 1.20f, 1.55f, 0.50f }
};
static const uint8_t BAND_COUNT = sizeof(BAND_CFGS) / sizeof(BAND_CFGS[0]);

BandResult results[BAND_COUNT];

BandTrack bandTracks[BAND_COUNT];

// ====================== GLOBALS ======================
float vReal[SAMPLES];
float vImag[SAMPLES];
ArduinoFFT<float> FFT(vReal, vImag, SAMPLES, SAMPLING_FREQ);

enum ParseState : uint8_t { WAIT_HEADER, READ_LEN, READ_DATA, READ_TAIL };
ParseState pState = WAIT_HEADER;

uint32_t headerShift = 0;
uint8_t lenBytes[2] = {0, 0};
uint8_t lenPos = 0;
uint16_t pLen = 0;
uint8_t pPayload[64];
uint16_t payloadPos = 0;
uint8_t tailBytes[4];
uint8_t tailPos = 0;
```

```
bool detected = false;
uint8_t resultByte = 0;
bool movingPresent = false;
bool staticPresent = false;

float rawDistM = 0.0f;
float filtDistM = 0.0f;
bool hasDistance = false;
uint16_t gateEnergies[16] = {0};
uint16_t targetEnergy = 0;

unsigned long lastFrameMs = 0;
unsigned long lastGoodFrameMs = 0;
unsigned long lastModeCmdMs = 0;
volatile unsigned long radarByteCount = 0;
volatile unsigned long radarFrameCount = 0;
unsigned long lastRadarDiagMs = 0;
unsigned long lastRadarDiagByteCount = 0;
unsigned long lastRadarDiagFrameCount = 0;

float sampleRing[SAMPLES];
uint16_t sampleWriteIndex = 0;
uint16_t sampleCount = 0;
unsigned long lastSampleMs = 0;
unsigned long lastAnalyzeMs = 0;
float lastSampleDistM = 0.0f;
unsigned long lastDistUpdateMs = 0;
double lastSampleValue = 0.0;
bool hasLastSampleValue = false;
float recentDistHist[DIST_STABLE_HISTORY_N] = {0.0f};
uint8_t recentDistWrite = 0;
uint8_t recentDistCount = 0;
unsigned long quietUntilMs = 0;

int lastBpm = -1;
bool bpmValid = false;
```

```
double lastSnr = 0.0;
double lastPeak = 0.0;
double lastVar = 0.0;
bool movingNow = false;
unsigned long lastBpmGoodMs = 0;
unsigned long bandLockUntilMs = 0;

// anti-drift voters
uint8_t upVotes = 0;
uint8_t downVotes = 0;

// classifier history
int clsBpmHist[CLASS_HIST_N] = {0};
uint8_t clsBandHist[CLASS_HIST_N] = {BAND_UNKNOWN};
uint8_t clsWrite = 0;
uint8_t clsCount = 0;

// classifier state
uint8_t classifierBand = BAND_UNKNOWN;
uint8_t pendingClassifierBand = BAND_UNKNOWN;
uint8_t classifierSwitchVotes = 0;

const char* activeBandName = "Unknown";
bool distTrackerInit = false;
float distTrackM = 0.0f;
float distTrackVelMps = 0.0f;
bool bpmTrackerInit = false;
float bpmKalmanX = 0.0f;
float bpmKalmanP = 25.0f;
int8_t lockedGate = -1;
uint8_t energySpikeStreak = 0;
bool motionLockout = false;
unsigned long motionLockoutUntilMs = 0;

// Cardiac detection globals
int lastHrBpm = -1;
bool hrValid = false;
```

```
double lastHrSnr = 0.0;
unsigned long lastHrGoodMs = 0;
unsigned long lastCardiacAnalyzeMs = 0;
// Add after lastCardiacAnalyzeMs:
float cardiacRing[CARDIAC_SAMPLES];
uint16_t cardiacWriteIndex = 0;
uint16_t cardiacCount = 0;
float cardiacVReal[CARDIAC_SAMPLES];
float cardiacVImag[CARDIAC_SAMPLES];
ArduinoFFT<float> cardiacFFT(cardiacVReal, cardiacVImag, CARDIAC_SAMPLES, CARDIAC_FREQ);
float hrKalmanX = 0.0f;
float hrKalmanP = 100.0f;
bool  hrKalmanInit = false;
int hrMedianHist[HR_MEDIAN_N] = {0};
uint8_t hrMedianWrite = 0;
uint8_t hrMedianCount = 0;
uint8_t hrSearchBand = BAND_UNKNOWN;
uint8_t pendingHrSearchBand = BAND_UNKNOWN;
uint8_t hrBandSwitchVotes = 0;
int hrBootstrapHist[HR_BOOTSTRAP_N] = {0};
uint8_t hrBootstrapWrite = 0;
uint8_t hrBootstrapCount = 0;
uint8_t distJumpStreak = 0;
float lockedSubjectDistM = 0.0f;
bool subjectDistLocked = false;
unsigned long subjectDistDivergeMs = 0;  // when distance first exceeded lock tolerance
#define SUBJECT_LOCK_RELEASE_MS  4000UL  // release lock after 4s outside tolerance

RadarLcd1602 statusLcd;

// ====================== HELPERS ======================
static inline double median3(double a, double b, double c) {
  if (a > b) { double t = a; a = b; b = t; }
  if (b > c) { double t = b; b = c; c = t; }
  if (a > b) { double t = a; a = b; b = t; }
  return b;
}
```

```
void sendReportModeCmd() {
  RADAR_SERIAL_PORT.write(CMD_SET_REPORT_MODE, sizeof(CMD_SET_REPORT_MODE));
  RADAR_SERIAL_PORT.flush();
  lastModeCmdMs = millis();
}

void radar2RecoverLink() {
    resetParser();
    for (int i = 0; i < 8; i++) {     // more attempts
        sendReportModeCmd();
        delay(25);
        radar2LoopStep();             // service UART while waiting
    }
    lastGoodFrameMs = millis();
    lastFrameMs = lastGoodFrameMs;
    Serial.println("Radar report mode command sent aggressively");
}

void radar2ShowStatus(const char *line0, const char *line1) {
  statusLcd.clear();
  statusLcd.printFixed(0, 0, line0 ? line0 : "");
  statusLcd.printFixed(0, 1, line1 ? line1 : "");
}

void radar2RestartUartOnly() {
#if defined(ARDUINO_UNOR4_WIFI)
  RADAR_SERIAL_PORT.begin(115200);
#else
  RADAR_SERIAL_PORT.begin(115200, SERIAL_8N1, RADAR_RX_PIN, RADAR_TX_PIN);
#endif
  Serial.println("Radar UART restarted");
}

void resetParser() {
  pState = WAIT_HEADER;
  headerShift = 0;
```

```
  lenPos = 0;
  pLen = 0;
  payloadPos = 0;
  tailPos = 0;
}

void metersToFeetInches(float meters, int &feetOut, float &inchesOut) {
  float totalInches = meters * 39.3701f;
  feetOut = (int)(totalInches / 12.0f);
  inchesOut = totalInches - (feetOut * 12.0f);
}

const char* modeLabel() {
  if (resultByte == 0x03) return "Both";
  if (resultByte == 0x02) return "Stationary";
  if (resultByte == 0x01) return "Moving";
  return "None";
}

const char* ageHintForBpm(int bpm) {
  static char hint[64];
  hint[0] = '\0';

  auto addBand = [&](const char* name, int lo, int hi) {
    if (bpm >= lo && bpm <= hi) {
      if (hint[0] != '\0') strcat(hint, " | ");
      strcat(hint, name);
    }
  };

  addBand("Infant", 30, 60);
  addBand("Child", 18, 40);
  addBand("Adult", 12, 20);

  if (hint[0] == '\0') strcpy(hint, "OutOfRange");
  return hint;
}
```

```
uint16_t weightedEnergyForDistance(uint16_t distCmRef) {
  int gate = constrain((int)(distCmRef / 70), 0, 15);
  auto gateEnergyAt = [&](int g) -> uint16_t {
    g = constrain(g, 0, 15);
    return gateEnergies[g];
  };

  uint16_t e0 = gateEnergyAt(gate);
  uint16_t e1 = gateEnergyAt(gate - 1);
  uint16_t e2 = gateEnergyAt(gate + 1);

  if (distCmRef > 300) {
    uint16_t e3 = gateEnergyAt(gate + 2);
    uint16_t e4 = gateEnergyAt(gate - 2);
    return (uint16_t)(0.40f * e0 + 0.20f * e1 + 0.20f * e2 + 0.10f * e3 + 0.10f * e4);
  }

  return (uint16_t)(0.60f * e0 + 0.20f * e1 + 0.20f * e2);
}

uint16_t nearestStrongGateDistanceCm(uint16_t fallbackDistCm) {
  uint16_t maxEnergy = 0;
  for (uint8_t g = 0; g < 16; g++) {
    if (gateEnergies[g] > maxEnergy) maxEnergy = gateEnergies[g];
  }

  uint16_t gateThreshold = max<uint16_t>(6, maxEnergy / 4);
  if (lockedGate >= 0 && lockedGate < 16) {
    uint16_t lockedEnergy = gateEnergies[lockedGate];
    if (lockedEnergy >= (uint16_t)(gateThreshold * GATE_KEEP_RATIO)) {
      uint16_t gateCenterCm = (uint16_t)(lockedGate * 70 + 35);
      int gLo = max(0, (int)lockedGate - 1);
      int gHi = min(15, (int)lockedGate + 1);
      uint32_t weightedPos = 0;
      uint32_t weightSum = 0;
      for (int k = gLo; k <= gHi; k++) {
```

```
      uint16_t e = gateEnergies[k];
      if (e == 0) continue;
      uint16_t centerCm = (uint16_t)(k * 70 + 35);
      weightedPos += (uint32_t)e * centerCm;
      weightSum += e;
    }
    if (weightSum > 0) {
      uint16_t refinedCm = (uint16_t)(weightedPos / weightSum);
      return (uint16_t)(0.65f * refinedCm + 0.35f * fallbackDistCm);
    }
    return gateCenterCm;
  }
}

for (uint8_t g = 0; g < 16; g++) {
  uint16_t gateCenterCm = (uint16_t)(g * 70 + 35);
  if (gateCenterCm < (uint16_t)(DETECT_MIN_DIST_M * 100.0f)) continue;
  if (gateCenterCm > (uint16_t)(DETECT_MAX_DIST_M * 100.0f)) break;
  if (gateEnergies[g] >= gateThreshold) {
    lockedGate = g;
    int gLo = max(0, (int)g - 1);
    int gHi = min(15, (int)g + 1);
    uint32_t weightedPos = 0;
    uint32_t weightSum = 0;

    for (int k = gLo; k <= gHi; k++) {
      uint16_t e = gateEnergies[k];
      if (e == 0) continue;
      uint16_t centerCm = (uint16_t)(k * 70 + 35);
      weightedPos += (uint32_t)e * centerCm;
      weightSum += e;
    }

    if (weightSum > 0) {
      uint16_t refinedCm = (uint16_t)(weightedPos / weightSum);
      // Blend the refined nearest-gate estimate with the radar-reported
      // distance so the display stays on the closest target but is less quantized.
```

```
        return (uint16_t)(0.65f * refinedCm + 0.35f * fallbackDistCm);
      }

      return gateCenterCm;
    }
  }

  lockedGate = -1;
  return fallbackDistCm;
}

void resetHrSmoother() {
    hrKalmanInit = false;
    hrKalmanX = 0.0f;
    hrKalmanP = 100.0f;
    hrMedianWrite = 0;
    hrMedianCount = 0;
    hrSearchBand = BAND_UNKNOWN;
    pendingHrSearchBand = BAND_UNKNOWN;
    hrBandSwitchVotes = 0;
    hrBootstrapWrite = 0;
    hrBootstrapCount = 0;
    for (uint8_t i = 0; i < HR_MEDIAN_N; i++) hrMedianHist[i] = 0;
    for (uint8_t i = 0; i < HR_BOOTSTRAP_N; i++) hrBootstrapHist[i] = 0;
}

uint8_t updateHrSearchBand() {
    uint8_t proposed = (classifierBand < BAND_COUNT) ? classifierBand : BAND_UNKNOWN;

    if (hrSearchBand == BAND_UNKNOWN) {
        if (proposed != BAND_UNKNOWN) hrSearchBand = proposed;
        return (hrSearchBand == BAND_UNKNOWN) ? BAND_ADULT : hrSearchBand;
    }

    if (proposed == BAND_UNKNOWN || proposed == hrSearchBand) {
        pendingHrSearchBand = BAND_UNKNOWN;
        hrBandSwitchVotes = 0;
```

```
        return hrSearchBand;
    }

    if (pendingHrSearchBand == proposed) {
        hrBandSwitchVotes++;
    } else {
        pendingHrSearchBand = proposed;
        hrBandSwitchVotes = 1;
    }

    if (hrBandSwitchVotes >= HR_BAND_SWITCH_VOTES) {
        hrSearchBand = proposed;
        pendingHrSearchBand = BAND_UNKNOWN;
        hrBandSwitchVotes = 0;
    }

    return hrSearchBand;
}

void clearSamples() {
  sampleCount = 0;
  sampleWriteIndex = 0;
  hasLastSampleValue = false;
  cardiacCount = 0;          // add
  cardiacWriteIndex = 0;     // add
  resetHrSmoother();
  lastHrBpm = -1;
  hrValid = false;
}

void clearBreathingSamples() {
    sampleCount = 0;
    sampleWriteIndex = 0;
    hasLastSampleValue = false;
    // cardiac buffer intentionally preserved
}
//===========SAMPLE MANAGEMENT===============
```

```
void pushSample(float v) {
  sampleRing[sampleWriteIndex] = v;
  sampleWriteIndex = (sampleWriteIndex + 1) % SAMPLES;
  if (sampleCount < SAMPLES) sampleCount++;
}

void pushCardiacSample(float v) {
    cardiacRing[cardiacWriteIndex] = v;
    cardiacWriteIndex = (cardiacWriteIndex + 1) % CARDIAC_SAMPLES;
    if (cardiacCount < CARDIAC_SAMPLES) cardiacCount++;
}

int hrMedianFilter(int measured) {
    hrMedianHist[hrMedianWrite] = measured;
    hrMedianWrite = (hrMedianWrite + 1) % HR_MEDIAN_N;
    if (hrMedianCount < HR_MEDIAN_N) hrMedianCount++;
    if (hrMedianCount < HR_MEDIAN_N) return measured;

    int a = hrMedianHist[0];
    int b = hrMedianHist[1];
    int c = hrMedianHist[2];
    if (a > b) { int t = a; a = b; b = t; }
    if (b > c) { int t = b; b = c; c = t; }
    if (a > b) { int t = a; a = b; b = t; }
    return b;
}

bool hrBootstrapReady(int candidateHr, double snr, int &seedHrOut) {
    if (snr < HR_BOOTSTRAP_SNR_MIN) return false;

    hrBootstrapHist[hrBootstrapWrite] = candidateHr;
    hrBootstrapWrite = (hrBootstrapWrite + 1) % HR_BOOTSTRAP_N;
    if (hrBootstrapCount < HR_BOOTSTRAP_N) hrBootstrapCount++;
    if (hrBootstrapCount < HR_BOOTSTRAP_N) return false;

    int hMin = hrBootstrapHist[0];
    int hMax = hrBootstrapHist[0];
```

```
    for (uint8_t i = 1; i < HR_BOOTSTRAP_N; i++) {
        if (hrBootstrapHist[i] < hMin) hMin = hrBootstrapHist[i];
        if (hrBootstrapHist[i] > hMax) hMax = hrBootstrapHist[i];
    }

    if ((hMax - hMin) > HR_BOOTSTRAP_SPAN_BPM) {
        hrBootstrapCount = 0;
        hrBootstrapWrite = 0;
        for (uint8_t i = 0; i < HR_BOOTSTRAP_N; i++) hrBootstrapHist[i] = 0;
        return false;
    }

    seedHrOut = median3(hrBootstrapHist[0], hrBootstrapHist[1], hrBootstrapHist[2]);
    return true;
}

int updateHrKalman(int measured, double snr) {
    measured = hrMedianFilter(measured);

    float R = HR_KALMAN_R;
    if (snr > 1.0) R = max(4.0f, (float)(HR_KALMAN_R / min(3.0, snr)));

    if (!hrKalmanInit) {
        hrKalmanX = (float)measured;
        hrKalmanP = R;
        hrKalmanInit = true;
        return measured;
    }

    int predicted = (int)roundf(hrKalmanX);
    int delta = measured - predicted;
    int maxStep = (snr >= HR_FAST_STEP_SNR) ? HR_MAX_STEP_BPM * 2 : HR_MAX_STEP_BPM;
    if (abs(delta) > HR_OUTLIER_STEP_BPM && snr < HR_FAST_STEP_SNR) {
        measured = predicted + ((delta > 0) ? HR_MAX_STEP_BPM : -HR_MAX_DOWN_STEP_BPM);
    } else if (delta < -HR_MAX_DOWN_STEP_BPM && snr < HR_FAST_STEP_SNR) {
        measured = predicted - HR_MAX_DOWN_STEP_BPM;
    } else if (abs(delta) > maxStep) {
```

```
      measured = predicted + ((delta > 0) ? maxStep : -maxStep);
    }

    hrKalmanP += HR_KALMAN_Q;
    float stabilityBonus = (hrKalmanP < 5.0f) ? 0.7f : 1.0f;
    float K = hrKalmanP / (hrKalmanP + R * stabilityBonus);
    hrKalmanX = hrKalmanX + K * ((float)measured - hrKalmanX);
    hrKalmanP = (1.0f - K) * hrKalmanP;
    return (int)roundf(hrKalmanX);
}

void pushRecentDistance(float d) {
  recentDistHist[recentDistWrite] = d;
  recentDistWrite = (recentDistWrite + 1) % DIST_STABLE_HISTORY_N;
  if (recentDistCount < DIST_STABLE_HISTORY_N) recentDistCount++;
}

bool distanceLooksStable() {
  if (recentDistCount < DIST_STABLE_HISTORY_N) return false;

  float dMin = recentDistHist[0];
  float dMax = recentDistHist[0];
  for (uint8_t i = 1; i < recentDistCount; i++) {
    if (recentDistHist[i] < dMin) dMin = recentDistHist[i];
    if (recentDistHist[i] > dMax) dMax = recentDistHist[i];
  }
  return (dMax - dMin) <= DIST_STABLE_SPAN_M;
}

// ==========Kalman ========================
void updateDistanceAlphaBeta(float measuredDistM, float dt) {
  if (!distTrackerInit) {
    distTrackM = measuredDistM;
    distTrackVelMps = 0.0f;
    distTrackerInit = true;
    return;
  }
```

```
  float predicted = distTrackM + distTrackVelMps * dt;
  float residual = measuredDistM - predicted;
  distTrackM = predicted + DIST_ALPHA_BETA_ALPHA * residual;
  if (dt > 1e-3f) {
    distTrackVelMps += (DIST_ALPHA_BETA_BETA * residual) / dt;
  }
}

int updateBpmKalman(int measuredBpm, double snr) {
  float R = BPM_KALMAN_R_BASE;
  if (snr > 0.5) {
    R = max(1.2f, (float)(BPM_KALMAN_R_BASE / min(4.0, snr)));
  }

  if (!bpmTrackerInit) {
    bpmKalmanX = (float)measuredBpm;
    bpmKalmanP = R;
    bpmTrackerInit = true;
    return measuredBpm;
  }

  bpmKalmanP += BPM_KALMAN_Q;
  float K = bpmKalmanP / (bpmKalmanP + R);
  bpmKalmanX = bpmKalmanX + K * ((float)measuredBpm - bpmKalmanX);
  bpmKalmanP = (1.0f - K) * bpmKalmanP;
  int result = (int)roundf(bpmKalmanX);
  // Clamp to a sane physiological floor — prevents Kalman drift below Adult minimum
  if (result < 9) {
     bpmKalmanX = 9.0f;
     result = 9;
  }
  return result;
}

void updateBandTrack(uint8_t idx, const BandResult &r) {
  BandTrack &t = bandTracks[idx];
```

```
  if (!r.valid) {
    t.confidence *= 0.85f;
    return;
  }

  if (!t.initialized) {
    t.initialized = true;
    t.filteredBpm = (float)r.bpm;
    t.confidence = min(1.0f, (float)(0.35 + 0.12 * r.snr));
    return;
  }

  float diff = fabsf((float)r.bpm - t.filteredBpm);
  bool outlier = diff > 8.0f && r.snr < 3.2;
  if (outlier) {
    t.confidence *= 0.82f;
    return;
  }

  float alpha = (r.snr >= 3.0) ? 0.55f : 0.35f;
  t.filteredBpm = t.filteredBpm + alpha * ((float)r.bpm - t.filteredBpm);
  t.confidence = min(1.0f, t.confidence + (float)(0.10 + 0.05 * min(4.0, r.snr)));
}

bool bandPlausibleAtDistance(uint8_t band, float distM, uint16_t energy) {
    float expectedEnergyFloor;
    if (distM < 0.80f)       expectedEnergyFloor = 80.0f;
    else if (distM < 1.50f)  expectedEnergyFloor = 30.0f;
    else if (distM < 2.50f)  expectedEnergyFloor = 10.0f;
    else                     expectedEnergyFloor = 5.0f;

    if ((float)energy < expectedEnergyFloor) return false;
    if (band == BAND_INFANT && distM < 0.80f) return false;
    if (band == BAND_INFANT && distM < 1.20f && (float)energy < 300.0f) return false;
    if (band == BAND_INFANT && distM > 1.50f) return false;
    return true;
}
```

```
void copyAndFilterSamplesToFFT() {
  uint16_t start = (sampleCount == SAMPLES) ? sampleWriteIndex : 0;

  for (uint16_t i = 0; i < SAMPLES; i++) {
    uint16_t idx = (start + i) % SAMPLES;
    vReal[i] = sampleRing[idx];
    vImag[i] = 0.0;
  }

  for (uint16_t i = 1; i < SAMPLES - 1; i++) {
    double m = median3(vReal[i - 1], vReal[i], vReal[i + 1]);
    if (fabs(vReal[i] - m) > DESPIKE_DELTA) vReal[i] = m;
  }

  static double tmp[SAMPLES];
  tmp[0] = vReal[0];
  for (uint16_t i = 1; i < SAMPLES - 1; i++) {
    tmp[i] = (vReal[i - 1] + vReal[i] + vReal[i + 1]) / 3.0;
  }
  tmp[SAMPLES - 1] = vReal[SAMPLES - 1];
  for (uint16_t i = 0; i < SAMPLES; i++) vReal[i] = tmp[i];
}

void classifierPush(int bpm, uint8_t band) {
  clsBpmHist[clsWrite] = bpm;
  clsBandHist[clsWrite] = band;
  clsWrite = (clsWrite + 1) % CLASS_HIST_N;
  if (clsCount < CLASS_HIST_N) clsCount++;
}

int medianFromHistory() {
  int tmp[CLASS_HIST_N];
  int n = 0;
  for (uint8_t i = 0; i < clsCount; i++) {
    if (clsBpmHist[i] > 0) tmp[n++] = clsBpmHist[i];
  }
```

```
  if (n == 0) return -1;

  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (tmp[j] < tmp[i]) { int t = tmp[i]; tmp[i] = tmp[j]; tmp[j] = t; }
    }
  }
  return tmp[n / 2];
}

uint8_t classifyBand() {
  if (clsCount < CLASS_MIN_SAMPLES) {
    if (classifierBand == BAND_UNKNOWN) classifierBand = BAND_ADULT;
    return classifierBand;
  }

  int medBpm = medianFromHistory();
  if (medBpm < 0) return classifierBand == BAND_UNKNOWN ? BAND_ADULT : classifierBand;

  uint8_t proposed = BAND_ADULT;
  if (medBpm >= 44) proposed = BAND_INFANT;
  else if (medBpm >= 30) proposed = BAND_CHILD;

  if (classifierBand == BAND_UNKNOWN) {
    classifierBand = BAND_ADULT;
    pendingClassifierBand = BAND_UNKNOWN;
    classifierSwitchVotes = 0;
    return classifierBand;
  }

  if (proposed != classifierBand) {
    if (pendingClassifierBand == proposed) classifierSwitchVotes++;
    else {
      pendingClassifierBand = proposed;
      classifierSwitchVotes = 1;
    }
```

```
    if (classifierSwitchVotes >= CLASS_SWITCH_VOTES) {
      classifierBand = proposed;
      pendingClassifierBand = BAND_UNKNOWN;
      classifierSwitchVotes = 0;
      upVotes = 0;
      downVotes = 0;
    }
  } else {
    pendingClassifierBand = BAND_UNKNOWN;
    classifierSwitchVotes = 0;
  }

  return classifierBand;
}

// ====================== BAND EVAL ======================
BandResult evalBand(const BandCfg& cfg) {
  BandResult r;
  r.valid = false;
  r.bpm = -1;
  r.peakMag = 0.0;
  r.snr = 0.0;
  r.score = 0.0;
  r.peakBin = -1;

  int lowBin = max(1, (int)(cfg.fMinHz * SAMPLES / SAMPLING_FREQ));
  int highBin = min(SAMPLES / 2 - 2, (int)(cfg.fMaxHz * SAMPLES / SAMPLING_FREQ));
  if (highBin <= lowBin + 2) return r;

  int peakBin = lowBin;
  double peakMag = 0.0;
  for (int i = lowBin; i <= highBin; i++) {
    if (vReal[i] > peakMag) {
      peakMag = vReal[i];
      peakBin = i;
    }
  }
```

```
if (peakMag < 1e-9) return r;

// Check both half-harmonic (subharmonic) and double-harmonic
int halfBin = peakBin / 2;
int doubleBin = peakBin * 2;

// Prefer subharmonic if it's strong — likely the true fundamental
if (halfBin >= lowBin && halfBin <= highBin &&
    vReal[halfBin] > 0.50 * peakMag) {
    peakBin = halfBin;
    peakMag = vReal[halfBin];
}

// Suppress double-harmonic — if the bin above is twice our peak bin
// and stronger, we're looking at a harmonic of a lower rate
if (doubleBin <= highBin && vReal[doubleBin] > 1.8 * peakMag) {
    // The real fundamental is below our band — flag invalid
    BandResult r;
    r.valid = false;
    r.bpm = -1;
    r.peakMag = 0; r.snr = 0; r.score = 0; r.peakBin = -1;
    return r;
}

double noiseSum = 0.0;
int noiseN = 0;
for (int i = lowBin; i <= highBin; i++) {
  if (abs(i - peakBin) <= 1) continue;
  noiseSum += vReal[i];
  noiseN++;
}
double noiseFloor = (noiseN > 0) ? (noiseSum / noiseN) : 1.0;
if (noiseFloor < 1e-6) noiseFloor = 1e-6;
double snr = peakMag / noiseFloor;

double alpha = vReal[peakBin - 1];
double beta  = vReal[peakBin];
```

```
  double gamma = vReal[peakBin + 1];
  double denom = alpha - 2.0 * beta + gamma;
  double delta = (fabs(denom) > 1e-9) ? (0.5 * (alpha - gamma) / denom) : 0.0;
  if (delta > 1.0) delta = 1.0;
  if (delta < -1.0) delta = -1.0;

  double interpBin = peakBin + delta;
  double freqHz = interpBin * SAMPLING_FREQ / SAMPLES;
  int bpm = (int)round(freqHz * 60.0);

  float snrNeed = (bpm > cfg.restMax) ? cfg.snrHigh : cfg.snrBase;
  bool valid = (bpm >= cfg.bpmMin &&
                bpm <= cfg.bpmMax &&
                peakMag >= cfg.peakMin &&
                snr >= snrNeed);

  r.valid = valid;
  r.bpm = bpm;
  r.peakMag = peakMag;
  r.snr = snr;
  r.score = valid ? (snr * peakMag) : 0.0;
  r.peakBin = peakBin;
  return r;
}

// ====================== FRAME PARSER ======================
void processFrame(const uint8_t *payload, uint16_t len) {
  if (len < 35) return;

  resultByte = payload[0];
  movingPresent = (resultByte == 0x01 || resultByte == 0x03);
  staticPresent = (resultByte == 0x02 || resultByte == 0x03);
  detected = (resultByte != 0x00);
  if (!detected) return;

  uint16_t distCm = (uint16_t)(payload[1] | (payload[2] << 8));
  float newDistM = distCm / 100.0f;
```

```
  if (newDistM < DETECT_MIN_DIST_M || newDistM > DETECT_MAX_DIST_M) return;

  for (int g = 0; g < 16; g++) {
    int off = 3 + g * 2;
    gateEnergies[g] = (uint16_t)(payload[off] | (payload[off + 1] << 8));
  }

  distCm = nearestStrongGateDistanceCm(distCm);
  newDistM = distCm / 100.0f;

  unsigned long nowMs = millis();
  float dt = (lastDistUpdateMs == 0) ? (SAMPLE_PERIOD_MS / 1000.0f)
                                     : ((nowMs - lastDistUpdateMs) / 1000.0f);
  if (dt < 0.01f) dt = 0.01f;
  lastDistUpdateMs = nowMs;
  updateDistanceAlphaBeta(newDistM, dt);
  newDistM = distTrackM;

  rawDistM = newDistM;
  lastFrameMs = millis();
  lastGoodFrameMs = lastFrameMs;


if (!hasDistance) {
    filtDistM = newDistM;
    hasDistance = true;
    distTrackM = newDistM;
    distTrackVelMps = 0.0f;
} else {
    // Strong outlier rejection when we have a subject lock
    float outlierThreshold = subjectDistLocked ? 0.40f : DIST_OUTLIER_STEP_M;

    if (fabsf(newDistM - filtDistM) > outlierThreshold) {
        // Big jump — trust the alpha-beta prediction more, blend lightly
        filtDistM = 0.75f * filtDistM + 0.25f * newDistM;
    } else {
        // Normal case — use your existing alpha-beta filter (already in code)
```

```
      // It already does prediction + correction, which is better than plain EMA
      updateDistanceAlphaBeta(newDistM, dt);
      filtDistM = distTrackM;              // take the smoothed value from tracker
    }
  }

  uint16_t gateRefCm = (uint16_t)(filtDistM * 100.0f);
  targetEnergy = weightedEnergyForDistance(gateRefCm);
  pushRecentDistance(filtDistM);
}

void parseRawByte(uint8_t b) {
  switch (pState) {
    case WAIT_HEADER:
      headerShift = (headerShift << 8) | b;
      if (headerShift == 0xF4F3F2F1UL) {
        pState = READ_LEN;
        lenPos = 0;
      }
      break;

    case READ_LEN:
      lenBytes[lenPos++] = b;
      if (lenPos >= 2) {
        pLen = (uint16_t)(lenBytes[0] | (lenBytes[1] << 8));
        if (pLen == 35) {
          payloadPos = 0;
          pState = READ_DATA;
        } else {
          resetParser();
        }
      }
      break;

    case READ_DATA:
      pPayload[payloadPos++] = b;
      if (payloadPos >= pLen) {
```

```
        tailPos = 0;
        pState = READ_TAIL;
      }
      break;

    case READ_TAIL:
      tailBytes[tailPos++] = b;
      if (tailPos >= 4) {
        if (tailBytes[0] == 0xF8 && tailBytes[1] == 0xF7 &&
            tailBytes[2] == 0xF6 && tailBytes[3] == 0xF5) {
          radarFrameCount++;
          processFrame(pPayload, pLen);
        }
        resetParser();
      }
      break;
  }
}

// ====================== SAMPLING UPDATE ======================
void updateSampling() {
  if (!detected || !hasDistance) return;
  if (filtDistM < BPM_MIN_DIST_M || filtDistM > BPM_MAX_DIST_M) return;

  unsigned long now = millis();
  if (now - lastSampleMs < SAMPLE_PERIOD_MS) return;
  lastSampleMs = now;

static float distWindow[5] = {0};
static uint8_t distWinIdx = 0;
static bool distWinFull = false;

distWindow[distWinIdx] = filtDistM;
distWinIdx = (distWinIdx + 1) % 5;
if (distWinIdx == 0) distWinFull = true;

if (distWinFull) {
```

```
    float dMin = distWindow[0], dMax = distWindow[0];
    for (uint8_t i = 1; i < 5; i++) {
      if (distWindow[i] < dMin) dMin = distWindow[i];
      if (distWindow[i] > dMax) dMax = distWindow[i];
    }
    if (dMax - dMin > 0.20f) {
      distJumpStreak++;
      if (distJumpStreak >= 2) {
        distJumpStreak = 0;
        clearBreathingSamples();
        motionLockout = true;
        motionLockoutUntilMs = now + 2000UL;
        movingNow = true;
        if (bpmValid || lastBpm > 0) {
          quietUntilMs = now + QUIET_SETTLE_MS;
        }
      }
      lastSampleDistM = filtDistM;
      return;
    } else {
      distJumpStreak = 0;
    }
}

lastSampleDistM = filtDistM;

// Subject distance lock — once BPM is acquired, only sample near that distance
// Adaptive lock tolerance — tighter once we have a stable BPM history
float lockTolerance = 0.30f;
if (bpmValid && clsCount >= 8) lockTolerance = 0.20f;
if (bpmValid && clsCount >= 12) lockTolerance = 0.15f;

if (subjectDistLocked && fabsf(filtDistM - lockedSubjectDistM) > lockTolerance) {
    if (subjectDistDivergeMs == 0) {
      subjectDistDivergeMs = now;
    } else if (now - subjectDistDivergeMs > SUBJECT_LOCK_RELEASE_MS) {
      // Subject has been outside lock range for too long — release lock
```

```
        subjectDistLocked = false;
        lockedSubjectDistM = 0.0f;
        subjectDistDivergeMs = 0;
        clearBreathingSamples();
        lastBpm = -1;
        bpmValid = false;
        resetHrSmoother();
        lastHrBpm = -1;
        hrValid = false;
        clsCount = 0;
        clsWrite = 0;
        classifierBand = BAND_UNKNOWN;
        bpmTrackerInit = false;
    }
    return;
} else {
    subjectDistDivergeMs = 0;  // reset timer when back in range
}

lastSampleDistM = filtDistM;

  float energyThresh;
  if (filtDistM > 4.0f)      energyThresh = 3.0f;
  else if (filtDistM > 2.0f) energyThresh = 15.0f;
  else                       energyThresh = 25.0f;
  if (targetEnergy < energyThresh) return;
  double sampleValue = log1p((double)targetEnergy);

  if (hasLastSampleValue && fabs(sampleValue - lastSampleValue) > ENERGY_SPIKE_LOG_DELTA) {
      energySpikeStreak++;
      if (energySpikeStreak >= 3) {
          energySpikeStreak = 0;
          clearBreathingSamples();
          motionLockout = true;
          motionLockoutUntilMs = now + 2000UL;
          movingNow = true;
          if (bpmValid || lastBpm > 0) {
```

```
            quietUntilMs = now + QUIET_SETTLE_MS;
          }
          lastSampleValue = sampleValue;
        } else {
          // Transient spike — clamp to last known good value
          pushSample(lastSampleValue);
        }
        return;
      }

      energySpikeStreak = 0;
      lastSampleValue = sampleValue;
      hasLastSampleValue = true;
      pushSample(sampleValue);
      pushCardiacSample(sampleValue);   // add this line

      // Force immediate analysis on first buffer fill
      static bool firstFillTriggered = false;
      if (!firstFillTriggered && sampleCount >= SAMPLES) {
        firstFillTriggered = true;
        lastAnalyzeMs = 0;   // expires the timer immediately
      }
    }

    // ====================== BREATHING ANALYSIS ======================
    void analyzeBreathingFast() {
      if (bpmValid) return;                    // full path already locked
      if (sampleCount < FAST_SAMPLES) return;      // not enough data yet
      if (sampleCount >= SAMPLES) return;          // full path will handle it

      static unsigned long lastFastMs = 0;
      unsigned long now = millis();
      if (now - lastFastMs < FAST_ANALYZE_PERIOD_MS) return;
      lastFastMs = now;

      // Copy the most recent FAST_SAMPLES into vReal, zero-pad to SAMPLES
      uint16_t start = (sampleWriteIndex >= FAST_SAMPLES)
```

```
                ? (sampleWriteIndex - FAST_SAMPLES)
                : (SAMPLES + sampleWriteIndex - FAST_SAMPLES);
for (int i = 0; i < SAMPLES; i++) {
  if (i < FAST_SAMPLES) {
    vReal[i] = sampleRing[(start + i) % SAMPLES];
  } else {
    vReal[i] = 0.0;
  }
  vImag[i] = 0.0;
}

// Despike
for (uint16_t i = 1; i < FAST_SAMPLES - 1; i++) {
  double m = median3(vReal[i-1], vReal[i], vReal[i+1]);
  if (fabs(vReal[i] - m) > DESPIKE_DELTA) vReal[i] = m;
}

// Mean-subtract only the live portion
double mean = 0.0;
for (int i = 0; i < FAST_SAMPLES; i++) mean += vReal[i];
mean /= FAST_SAMPLES;
for (int i = 0; i < FAST_SAMPLES; i++) vReal[i] -= mean;

double var = 0.0;
for (int i = 0; i < FAST_SAMPLES; i++) var += vReal[i] * vReal[i];
var /= FAST_SAMPLES;
float cardiacVarThresh = MOTION_VAR_THRESHOLD;
if (filtDistM < 0.80f) cardiacVarThresh += (0.80f - filtDistM) * 0.60f;
if (var > cardiacVarThresh) return;

FFT.windowing(FFT_WIN_TYP_HANN, FFT_FORWARD);
FFT.compute(FFT_FORWARD);
FFT.complexToMagnitude();

// Evaluate all bands but require higher SNR since window is short
int winner = -1;
double bestScore = -1.0;
```

```
  for (uint8_t i = 0; i < BAND_COUNT; i++) {
    BandResult r = evalBand(BAND_CFGS[i]);
    if (r.valid && r.snr >= FAST_SNR_MIN && r.score > bestScore) {
      bestScore = r.score;
      winner = i;
    }
  }

  if (winner < 0) return;

  BandResult br = evalBand(BAND_CFGS[winner]);
  lastBpm = updateBpmKalman(br.bpm, br.snr);
  lastSnr = br.snr;
  lastPeak = br.peakMag;
  activeBandName = BAND_CFGS[winner].name;
  classifierBand = (uint8_t)winner;
  bpmValid = true;
  lastBpmGoodMs = millis();
  bandLockUntilMs = millis() + BAND_LOCK_HOLD_MS;
}

void analyzeBreathing() {
  unsigned long now = millis();
  if (now - lastAnalyzeMs < ANALYZE_PERIOD_MS) return;
  lastAnalyzeMs = now;

  if (now < quietUntilMs) {
    movingNow = false;
    bpmValid = false;
    activeBandName = "Unknown";
    lastSnr = 0.0;
    return;
  }

  if (motionLockout) {
    if (millis() < motionLockoutUntilMs) {
        bpmValid = false;
```

```
        activeBandName = "Unknown";
        return;
      }
      motionLockout = false;
    }

    if (sampleCount < SAMPLES) {
      movingNow = false;
      bpmValid = false;
      activeBandName = "Unknown";
      lastSnr = 0.0;
      return;
    }

    copyAndFilterSamplesToFFT();

    double mean = 0.0;
    for (int i = 0; i < SAMPLES; i++) mean += vReal[i];
    mean /= SAMPLES;
    for (int i = 0; i < SAMPLES; i++) vReal[i] -= mean;

    double var = 0.0;
    for (int i = 0; i < SAMPLES; i++) var += vReal[i] * vReal[i];
    var /= SAMPLES;
    lastVar = var;

    float effectiveVarThresh = MOTION_VAR_THRESHOLD;
    if (filtDistM < 0.80f) effectiveVarThresh += (0.80f - filtDistM) * 0.60f;
    if (filtDistM > 3.0f)  effectiveVarThresh += (filtDistM - 3.0f) * 10.0f;

    if (var > effectiveVarThresh) {
      movingNow = true;
      bpmValid = false;
      activeBandName = "Unknown";
      return;
    }
    movingNow = false;
```

```
  FFT.windowing(FFT_WIN_TYP_HANN, FFT_FORWARD);
  FFT.compute(FFT_FORWARD);
  FFT.complexToMagnitude();

  int winner = -1;
  double bestScore = -1.0;
  for (uint8_t i = 0; i < BAND_COUNT; i++) {
    results[i] = evalBand(BAND_CFGS[i]);
    updateBandTrack(i, results[i]);

    // Distance-aware plausibility gate
    if (results[i].valid && !bandPlausibleAtDistance(i, filtDistM, targetEnergy)) {
        results[i].valid = false;
        results[i].score = 0.0;
        bandTracks[i].confidence *= 0.70f;  // penalize track confidence too
    }

    if (results[i].valid && results[i].score > bestScore) {
        bestScore = results[i].score;
        winner = i;
    }
  }

  // 1.5x harmonic correction — reading 20-25 BPM when true rate is 13-17
  if (results[BAND_ADULT].valid &&
    results[BAND_ADULT].bpm >= 20 &&
    results[BAND_ADULT].bpm <= 25 &&
    results[BAND_ADULT].snr < 3.5) {
    double candidateHz = (results[BAND_ADULT].bpm / 60.0) * (2.0 / 3.0);
    int fundamentalBin = (int)round(candidateHz * SAMPLES / SAMPLING_FREQ);
    fundamentalBin = constrain(fundamentalBin, 1, SAMPLES / 2 - 2);
    int fundamentalBpm = (int)round(candidateHz * 60.0);
    if (fundamentalBpm >= 11 && fundamentalBpm <= 18) {
        double fundamentalMag = vReal[fundamentalBin];
        if (fundamentalMag > 0.20 * results[BAND_ADULT].peakMag) {
            results[BAND_ADULT].bpm = fundamentalBpm;
```

```
            results[BAND_ADULT].peakMag = fundamentalMag;
            results[BAND_ADULT].score = results[BAND_ADULT].snr * fundamentalMag;
        }
    }
}

if (results[BAND_CHILD].valid &&
    results[BAND_CHILD].bpm >= 22 &&
    results[BAND_CHILD].bpm <= 30 &&
    results[BAND_CHILD].snr < 3.5) {
    double candidateHz = (results[BAND_CHILD].bpm / 60.0) * (2.0 / 3.0);
    int fundamentalBin = (int)round(candidateHz * SAMPLES / SAMPLING_FREQ);
    fundamentalBin = constrain(fundamentalBin, 1, SAMPLES / 2 - 2);
    int fundamentalBpm = (int)round(candidateHz * 60.0);
    if (fundamentalBpm >= 11 && fundamentalBpm <= 20) {
        double fundamentalMag = vReal[fundamentalBin];
        if (fundamentalMag > 0.20 * results[BAND_CHILD].peakMag) {
            if (!results[BAND_ADULT].valid ||
                fundamentalMag * results[BAND_CHILD].snr > results[BAND_ADULT].score) {
                results[BAND_ADULT].valid = true;
                results[BAND_ADULT].bpm = fundamentalBpm;
                results[BAND_ADULT].peakMag = fundamentalMag;
                results[BAND_ADULT].snr = results[BAND_CHILD].snr;
                results[BAND_ADULT].score = results[BAND_CHILD].snr * fundamentalMag;
                results[BAND_ADULT].peakBin = fundamentalBin;
                results[BAND_CHILD].valid = false;
                results[BAND_CHILD].score = 0.0;
            }
        }
    }
}

  int trackedWinner = -1;
  float trackedScore = -1.0f;
  for (uint8_t i = 0; i < BAND_COUNT; i++) {
    if (!bandTracks[i].initialized) continue;
    // Don't let a distance-implausible band win via track confidence alone
```

```
  if (!bandPlausibleAtDistance(i, filtDistM, targetEnergy)) continue;
  float score = bandTracks[i].confidence;
  if (results[i].valid) score += (float)(0.20 * min(4.0, results[i].snr));
  if (score > trackedScore) {
    trackedScore = score;
    trackedWinner = i;
  }
}

if (trackedWinner >= 0 && bandTracks[trackedWinner].confidence >= 0.55f) {
  winner = trackedWinner;
}

if (winner < 0) {
  bpmValid = (lastBpm > 0 && (millis() - lastBpmGoodMs) < BPM_HOLD_MS);
  if (!bpmValid) {
    activeBandName = "Unknown";
    lastSnr = 0.0;
  }
  return;
}

if (classifierBand < BAND_COUNT && now < bandLockUntilMs) {
  // Don't hold a lock on a band that's implausible at the current distance
  if (!bandPlausibleAtDistance(classifierBand, filtDistM, targetEnergy)) {
      bandLockUntilMs = 0;  // expire the lock immediately
  } else {
    BandResult lockedNow = results[classifierBand];
    if (lockedNow.valid) {
      bool keepAdultLock =
          (classifierBand == BAND_ADULT &&
           lockedNow.bpm >= 12 && lockedNow.bpm <= 20 &&
           lockedNow.snr >= ADULT_STAY_SNR);
      bool keepStrongLock = lockedNow.snr >= FAST_LOCK_SNR;
      if (keepAdultLock || keepStrongLock) {
        winner = classifierBand;
      }
```

```
      }
    }
  }

  if (results[winner].valid && results[winner].snr >= FAST_LOCK_SNR) {
    bool sameBand      = ((uint8_t)winner == classifierBand);
    bool lockExpired   = (now >= bandLockUntilMs);
    bool strongChallenger = (classifierBand < BAND_COUNT &&
                             results[winner].snr > results[classifierBand].snr + 1.5);
    if (sameBand || lockExpired || strongChallenger) {
        classifierBand = (uint8_t)winner;
        pendingClassifierBand = BAND_UNKNOWN;
        classifierSwitchVotes = 0;
        bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
    }
  }

  activeBandName = BAND_CFGS[winner].name;
  classifierPush(results[winner].bpm, (uint8_t)winner);
  uint8_t locked = classifyBand();

  BandResult br = results[locked];
  if (!br.valid) br = results[winner];

  if (!br.valid && bandTracks[locked].initialized && bandTracks[locked].confidence >= 0.60f) {
    br.valid = true;
    br.bpm = (int)roundf(bandTracks[locked].filteredBpm);
    br.snr = lastSnr;
    br.peakMag = lastPeak;
  } else if (!br.valid && bandTracks[winner].initialized && bandTracks[winner].confidence >= 0.60f) {
    br.valid = true;
    br.bpm = (int)roundf(bandTracks[winner].filteredBpm);
    br.snr = lastSnr;
    br.peakMag = lastPeak;
  }

  if (!br.valid) {
```

```
  bpmValid = (lastBpm > 0 && (millis() - lastBpmGoodMs) < BPM_HOLD_MS);
  if (!bpmValid) {
    activeBandName = "Unknown";
    lastSnr = 0.0;
  }
  return;
}

activeBandName = BAND_CFGS[locked].name;
int cand = br.bpm;
lastSnr = br.snr;
lastPeak = br.peakMag;

  // Prefer Adult band when reading is firmly in adult respiratory range
if (cand >= 11 && cand <= 20 && br.snr >= 2.0f &&
    classifierBand != BAND_ADULT &&
    results[BAND_ADULT].valid) {
    locked = BAND_ADULT;
    classifierBand = BAND_ADULT;
    bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
    pendingClassifierBand = BAND_UNKNOWN;
    classifierSwitchVotes = 0;
}

if (br.snr >= FAST_LOCK_SNR) {
  lastBpm = updateBpmKalman(cand, br.snr);
  bpmValid = true;
  lastBpmGoodMs = millis();
  bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
  upVotes = 0;
  downVotes = 0;
  // Set or refine subject lock
  if (!subjectDistLocked) {
      subjectDistLocked = true;
      lockedSubjectDistM = filtDistM;
  } else if (br.snr >= 4.0f) {
      lockedSubjectDistM = 0.85f * lockedSubjectDistM + 0.15f * filtDistM;
```

```
    }
    return;
  }

  // If we have a strong adult-range solution far away from a stale high BPM,
  // snap to it instead of ratcheting downward for many seconds.
  if (lastBpm > 0 &&
      locked == BAND_ADULT &&
      cand >= 12 && cand <= 20 &&
      abs(cand - lastBpm) >= 8 &&
      br.snr >= 2.2) {
    lastBpm = updateBpmKalman(cand, br.snr);
    bpmValid = true;
    lastBpmGoodMs = millis();
    classifierBand = BAND_ADULT;
    bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
    upVotes = 0;
    downVotes = 0;
    return;
  }

  if (lastBpm > 0 && cand > lastBpm + 6 && br.snr < 3.0) {
    bpmValid = (millis() - lastBpmGoodMs) < BPM_HOLD_MS;
    return;
  }

  static bool hasEverLocked = false;

 if (lastBpm <= 0 || !hasEverLocked) {
    if (br.snr < 2.50f) {
        bpmValid = false;
        return;
    }
    lastBpm = updateBpmKalman(cand, br.snr);
    bpmValid = true;
    hasEverLocked = true;
    if (!subjectDistLocked) {
```

```
      subjectDistLocked = true;
      lockedSubjectDistM = filtDistM;
    }
    lastBpmGoodMs = millis();
    bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
    upVotes = 0;
    downVotes = 0;
    return;
  }

  if (lastBpm > 0 && abs(cand - lastBpm) > 8 && br.snr < 5.0) {
    bpmValid = (millis() - lastBpmGoodMs) < BPM_HOLD_MS;
    return;
  }
  int d = cand - lastBpm;


  if (abs(d) <= 1) {
    upVotes = 0;
    downVotes = 0;
    bpmValid = true;
    lastBpmGoodMs = millis();
    return;
  }

  if (d > 0) {
    upVotes++;
    downVotes = 0;
    if (upVotes < UP_CONFIRM_N) {
      bpmValid = (millis() - lastBpmGoodMs) < BPM_HOLD_MS;
      return;
    }
    lastBpm += min(d, MAX_UP_STEP_BPM);
  } else {
    downVotes++;
    upVotes = 0;
    if (downVotes < DOWN_CONFIRM_N) {
```

```
      bpmValid = (millis() - lastBpmGoodMs) < BPM_HOLD_MS;
      return;
    }
    lastBpm -= min(-d, MAX_DOWN_STEP_BPM);
  }

  lastBpm = updateBpmKalman(lastBpm, br.snr);
  bpmValid = true;
  lastBpmGoodMs = millis();

  if (locked == BAND_ADULT && lastBpm >= 12 && lastBpm <= 20) {
    classifierBand = BAND_ADULT;
    bandLockUntilMs = now + BAND_LOCK_HOLD_MS;
  }
}

// ====================Cardiac Function====================
void analyzeCardiac() {
    if (CARDIAC_REQUIRE_BREATH_LOCK && (!bpmValid || lastBpm <= 0)) return;
    if (!detected || movingNow || motionLockout) return;
    if (cardiacCount < CARDIAC_SAMPLES) return;

    unsigned long now = millis();
    if (now - lastCardiacAnalyzeMs < CARDIAC_ANALYZE_MS) return;
    lastCardiacAnalyzeMs = now;

    uint8_t activeHrBand = updateHrSearchBand();

    // Select cardiac search parameters from the stabilized HR band lock.
    int cardiacMinBpm, cardiacMaxBpm;
    int cardiacNominalBpm;
    int lagSearchHalfWidth;
    int breathingMaxForHarmonicReject;
    switch (activeHrBand) {
        case BAND_INFANT:
            cardiacMinBpm = 100;
            cardiacMaxBpm = 160;
```

```
            cardiacNominalBpm = 130;
            lagSearchHalfWidth = 2;
            breathingMaxForHarmonicReject = 60;
            break;
        case BAND_CHILD:
            cardiacMinBpm = 70;
            cardiacMaxBpm = 130;
            cardiacNominalBpm = 95;
            lagSearchHalfWidth = 2;
            breathingMaxForHarmonicReject = 40;
            break;
        case BAND_ADULT:
        default:
            cardiacMinBpm = 65;
            cardiacMaxBpm = 105;
            cardiacNominalBpm = 80;
            lagSearchHalfWidth = 2;
            breathingMaxForHarmonicReject = 25;
            break;
    }

    int expectedHrBpm = cardiacNominalBpm;
    if (hrValid &&
        lastHrBpm >= cardiacMinBpm &&
        lastHrBpm <= cardiacMaxBpm &&
        (now - lastHrGoodMs) < CARDIAC_HOLD_MS) {
        expectedHrBpm = lastHrBpm;
        if (lastHrSnr >= 2.0) {
            lagSearchHalfWidth = 1;
        }
    }

    // Copy ring buffer into working array
    uint16_t start = cardiacWriteIndex;
    for (int i = 0; i < CARDIAC_SAMPLES; i++) {
        cardiacVReal[i] = cardiacRing[(start + i) % CARDIAC_SAMPLES];
        cardiacVImag[i] = 0.0;
```

```
}

// Despike
for (int i = 1; i < CARDIAC_SAMPLES - 1; i++) {
    double m = median3(cardiacVReal[i-1], cardiacVReal[i], cardiacVReal[i+1]);
    if (fabs(cardiacVReal[i] - m) > DESPIKE_DELTA) cardiacVReal[i] = m;
}

// Mean subtract
double mean = 0.0;
for (int i = 0; i < CARDIAC_SAMPLES; i++) mean += cardiacVReal[i];
mean /= CARDIAC_SAMPLES;
for (int i = 0; i < CARDIAC_SAMPLES; i++) cardiacVReal[i] -= mean;

// Variance gate
double var = 0.0;
for (int i = 0; i < CARDIAC_SAMPLES; i++) var += cardiacVReal[i] * cardiacVReal[i];
var /= CARDIAC_SAMPLES;
float varThresh = MOTION_VAR_THRESHOLD;
if (filtDistM < 0.80f) varThresh += (0.80f - filtDistM) * 0.60f;
if (var > varThresh) return;

// FFT to apply breathing notch
cardiacFFT.windowing(FFT_WIN_TYP_HANN, FFT_FORWARD);
cardiacFFT.compute(FFT_FORWARD);
cardiacFFT.complexToMagnitude();

// Apply notch on breathing harmonics in frequency domain
double breathHz = (lastBpm > 0) ? (lastBpm / 60.0) : 0.0;
if (breathHz > 0.0) {
    float snrFactor = (float)constrain(lastSnr, 1.0, 8.0);
    int baseNotch = constrain((int)round(5.0f - 0.4f * snrFactor), 2, 6);
    for (int harmonic = 1; harmonic <= 10; harmonic++) {
        double hHz = breathHz * harmonic;
        if (hHz > 3.0) break;
        int hBin = (int)round(hHz * CARDIAC_SAMPLES / CARDIAC_FREQ);
        int notchW = (harmonic <= 3) ? baseNotch :
```

```
                (harmonic <= 6) ? baseNotch + 2 : baseNotch + 1;
        int lo = max(1, hBin - notchW);
        int hi = min(CARDIAC_SAMPLES / 2 - 1, hBin + notchW);
        for (int b = lo; b <= hi; b++) {
            cardiacVReal[b] = 0.0;
            cardiacVImag[b] = 0.0;
        }
    }
}

int fftHrBpm = -1;
double fftPeakMag = 0.0;
int fftLowBin = max(1, (int)floor((cardiacMinBpm / 60.0f) * CARDIAC_SAMPLES / CARDIAC_FREQ));
int fftHighBin = min(CARDIAC_SAMPLES / 2 - 2,
                (int)ceil((cardiacMaxBpm / 60.0f) * CARDIAC_SAMPLES / CARDIAC_FREQ));
for (int b = fftLowBin; b <= fftHighBin; b++) {
    if (cardiacVReal[b] > fftPeakMag) {
        fftPeakMag = cardiacVReal[b];
        fftHrBpm = (int)round((b * CARDIAC_FREQ / CARDIAC_SAMPLES) * 60.0);
    }
}

// Inverse FFT to get notch-filtered time domain signal
cardiacFFT.compute(FFT_REVERSE);

// Autocorrelation lag search for cardiac period
int lagMin = (int)(CARDIAC_FREQ * 60.0 / (float)cardiacMaxBpm);
int lagMax = (int)(CARDIAC_FREQ * 60.0 / (float)cardiacMinBpm);
lagMin = max(lagMin, 2);
lagMax = min(lagMax, CARDIAC_SAMPLES / 3);

int expectedLag = (int)round(CARDIAC_FREQ * 60.0 / (float)expectedHrBpm);
expectedLag = constrain(expectedLag, lagMin, lagMax);
int lagSearchMin = max(lagMin, expectedLag - lagSearchHalfWidth);
int lagSearchMax = min(lagMax, expectedLag + lagSearchHalfWidth);

// Normalize signal power for autocorrelation
```

```
int acLen = CARDIAC_SAMPLES - lagMax;
double sigPower = 0.0;
for (int i = 0; i < acLen; i++) sigPower += cardiacVReal[i] * cardiacVReal[i];
if (sigPower < 1e-10) {
    hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
    return;
}

auto lagIsOnBreathingHarmonic = [&](int lag) -> bool {
    double lagBpm = CARDIAC_FREQ / lag * 60.0;
    if (breathHz <= 0.0 || lastBpm <= 0 || lastBpm > breathingMaxForHarmonicReject) {
        return false;
    }
    for (int h = 1; h <= 8; h++) {
        double harmonicBpm = breathHz * h * 60.0;
        if (fabs(lagBpm - harmonicBpm) <= 5.0) return true;
    }
    return false;
};

auto autocorrAtLag = [&](int lag) -> double {
    double ac = 0.0;
    for (int i = 0; i < acLen; i++) {
        ac += cardiacVReal[i] * cardiacVReal[i + lag];
    }
    return ac / sigPower;
};

auto findBestLagInWindow = [&](int searchMin, int searchMax, double &bestACOut, int &bestLagOut) {
    bestACOut = -1e9;
    bestLagOut = constrain(expectedLag, searchMin, searchMax);
    for (int lag = searchMin; lag <= searchMax; lag++) {
        if (lagIsOnBreathingHarmonic(lag)) continue;
        double ac = autocorrAtLag(lag);
        double distancePenalty = 1.0 - 0.04 * abs(lag - expectedLag);
        if (distancePenalty < 0.70) distancePenalty = 0.70;
        ac *= distancePenalty;
```

```
        if (ac > bestACOut) {
            bestACOut = ac;
            bestLagOut = lag;
        }
    }
};

// Search near the expected lag for this age band first, then fall back
// to the full band range if the narrow window is too weak.
double bestAC = -1e9;
int bestLag = expectedLag;
findBestLagInWindow(lagSearchMin, lagSearchMax, bestAC, bestLag);
if (!hrValid &&
    bestAC < 0.040 &&
    (lagSearchMin > lagMin || lagSearchMax < lagMax)) {
    findBestLagInWindow(lagMin, lagMax, bestAC, bestLag);
}

double expectedLagAC = -1e9;
if (!lagIsOnBreathingHarmonic(expectedLag)) {
    expectedLagAC = autocorrAtLag(expectedLag);
}

if (hrValid &&
    lastHrBpm >= cardiacMinBpm &&
    lastHrBpm <= cardiacMaxBpm &&
    (now - lastHrGoodMs) < CARDIAC_HOLD_MS &&
    expectedLagAC > 0.0 &&
    bestLag > expectedLag) {
    bool alias2x = abs(bestLag - 2 * expectedLag) <= 1;
    bool alias3to2 = abs(2 * bestLag - 3 * expectedLag) <= 1;
    if ((alias2x || alias3to2) && bestAC < expectedLagAC * HR_ALIAS_AC_RATIO) {
        bestLag = expectedLag;
        bestAC = expectedLagAC;
    }
}
```

```
for (int lag = bestLag * 2; lag <= lagMax; lag += bestLag) {
    double acHarm = 0.0;
    for (int i = 0; i < acLen; i++) {
        acHarm += cardiacVReal[i] * cardiacVReal[i + lag];
    }
    acHarm /= sigPower;
    if (acHarm > bestAC * 0.40 && acHarm > 0.015) {
        bestAC = acHarm;
        bestLag = lag;
        break;
    }
}

// Require minimum autocorrelation strength
if (bestAC < 0.040) {
    hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
    return;
}

// Confirm cardiac period by checking second and third harmonic lag
bool periodConfirmed = false;
int lag2 = bestLag * 2;
int lag3 = bestLag * 3;
if (lag2 <= CARDIAC_SAMPLES - lagMax - 1) {
    double ac2 = 0.0;
    for (int i = 0; i < acLen; i++) {
        ac2 += cardiacVReal[i] * cardiacVReal[i + lag2];
    }
    ac2 /= sigPower;
    if (ac2 > 0.015) periodConfirmed = true;
}
if (!periodConfirmed && lag3 <= CARDIAC_SAMPLES - lagMax - 1) {
    double ac3 = 0.0;
    for (int i = 0; i < acLen; i++) {
        ac3 += cardiacVReal[i] * cardiacVReal[i + lag3];
    }
    ac3 /= sigPower;
```

```
    if (ac3 > 0.010) periodConfirmed = true;
}

if (!periodConfirmed && bestAC < 0.08) {
    hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
    return;
}

// Edge lag wins are suspicious — apply extra scrutiny
if (bestLag == lagSearchMin) {
    double nextAC = 0.0;
    if (lagSearchMin + 1 <= lagSearchMax &&
        !lagIsOnBreathingHarmonic(lagSearchMin + 1)) {
        nextAC = autocorrAtLag(lagSearchMin + 1);
    }
    if (nextAC > bestAC * 0.30) {
        bestAC = nextAC;
        bestLag = lagSearchMin + 1;
    } else if (bestAC < 0.50) {
        hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
        return;
    }
}

// Parabolic interpolation to refine lag estimate
double lagRefined = bestLag;
if (bestLag > lagSearchMin && bestLag < lagSearchMax) {
    double acPrev = 0.0, acCurr = 0.0, acNext = 0.0;
    for (int i = 0; i < acLen; i++) {
        acPrev += cardiacVReal[i] * cardiacVReal[i + bestLag - 1];
        acCurr += cardiacVReal[i] * cardiacVReal[i + bestLag];
        acNext += cardiacVReal[i] * cardiacVReal[i + bestLag + 1];
    }
    double denom = acPrev - 2.0 * acCurr + acNext;
    if (fabs(denom) > 1e-10) {
        lagRefined = bestLag + 0.5 * (acPrev - acNext) / denom;
        lagRefined = constrain(lagRefined, (double)lagSearchMin, (double)lagSearchMax);
```

```
    }
}

double hrHz = CARDIAC_FREQ / lagRefined;
int hrCand = (int)round(hrHz * 60.0);

if (hrCand < cardiacMinBpm || hrCand > cardiacMaxBpm) {
    hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
    return;
}

if (fftHrBpm > 0 && abs(hrCand - fftHrBpm) > HR_FFT_AGREE_BPM) {
    if (abs(fftHrBpm - expectedHrBpm) < abs(hrCand - expectedHrBpm)) {
        hrCand = fftHrBpm;
    } else {
        hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
        return;
    }
}

if (hrValid &&
    lastHrBpm >= cardiacMinBpm &&
    lastHrBpm <= cardiacMaxBpm &&
    (now - lastHrGoodMs) < CARDIAC_HOLD_MS &&
    hrCand + HR_DROP_GUARD_BPM < lastHrBpm &&
    bestAC < expectedLagAC * HR_DROP_AC_RATIO) {
    lastHrSnr = constrain(max(lastHrSnr, expectedLagAC * 20.0), 0.5, 15.0);
    hrValid = true;
    return;
}

// HR must exceed breathing rate
if (lastBpm > 0 && hrCand <= lastBpm + 5) {
    hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
    return;
}
```

```
// Distance-aware plausibility
if (subjectDistLocked && lockedSubjectDistM < 1.20f) {
    if (hrCand > (int)(cardiacMaxBpm * 0.95f) && bestAC < 0.08) {
        hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
        return;
    }
}
if (subjectDistLocked && lockedSubjectDistM > 1.50f) {
    if (bestAC < 0.04) {
        hrValid = (!motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS));
        return;
    }
}

// Reset if Kalman is stuck in artifact range or candidate is far off
if (hrKalmanInit && (hrKalmanX > (float)cardiacMaxBpm ||
    (abs(hrCand - (int)hrKalmanX) > 25 && bestAC < 0.10))) {
    resetHrSmoother();
}

// Scale autocorrelation to SNR-like value for Kalman
double acSnr = bestAC * 20.0;
acSnr = constrain(acSnr, 0.5, 15.0);
lastHrSnr = acSnr;

if (!hrKalmanInit) {
    int seedHr = hrCand;
    if (!hrBootstrapReady(hrCand, acSnr, seedHr)) {
        hrValid = false;
        return;
    }
    hrCand = seedHr;
}

lastHrBpm = updateHrKalman(hrCand, acSnr);
hrValid = true;
lastHrGoodMs = now;
```

```
}
// ====================== STATUS ======================
void printStatus() {
  static unsigned long lastPrintMs = 0;
  unsigned long now = millis();
  bool freshFrame = (now - lastFrameMs) <= 1200UL;
  if (now - lastPrintMs < 1000) return;
  lastPrintMs = now;

  if (!freshFrame || (now - lastFrameMs > DETECT_HOLD_MS)) {
    bool wasDetected = detected;
    detected = false;
    movingPresent = false;
    staticPresent = false;
    bpmValid = false;
    hasDistance = false;
    if (wasDetected) {
        clsCount = 0;
        clsWrite = 0;
        classifierBand = BAND_UNKNOWN;
        pendingClassifierBand = BAND_UNKNOWN;
        classifierSwitchVotes = 0;
        bandLockUntilMs = 0;
        bpmTrackerInit = false;
        lastBpm = -1;
        subjectDistLocked = false;
        lockedSubjectDistM = 0.0f;
        subjectDistDivergeMs = 0;
    }
}

  if (!detected || !hasDistance) {
    Serial.println("Detected:NO Dist:-- BPM:--");
    statusLcd.showVitals(lastHrBpm,
                         false,
                         lastBpm,
                         false,
```

```
                    filtDistM,
                    false,
                    detected);
    return;
  }

  int ft = 0;
  float inch = 0.0f;
  metersToFeetInches(filtDistM, ft, inch);

  Serial.print("Detected:YES Dist:");
  Serial.print(filtDistM, 2);
  Serial.print(" m (");
  Serial.print(ft);
  Serial.print(" ft ");
  Serial.print(inch, 1);
  Serial.print(" in) Mode:");
  Serial.print(modeLabel());
  Serial.print(" Band:");
  Serial.print(activeBandName);
  if (subjectDistLocked) {
    Serial.print(" Lock:");
    Serial.print(lockedSubjectDistM, 2);
    Serial.print("m");
}
  Serial.print(' ');

  if (movingNow) {
    Serial.print("BPM:moving var=");
    Serial.print(lastVar, 3);
  } else if (bpmValid) {
    int displayBpm = lastBpm;
    // If locked to Adult band, clamp to physiological Adult range
    if (classifierBand == BAND_ADULT && displayBpm > 23) displayBpm = 23;
    if (classifierBand == BAND_CHILD && displayBpm > 40) displayBpm = 40;
    Serial.print("BPM:");
    Serial.print(max(11, displayBpm));
```

```
    Serial.print(" snr=");
    Serial.print(lastSnr, 2);
    Serial.print(" AgeHint:");
    Serial.print(ageHintForBpm(lastBpm));
  } else {
    Serial.print("BPM:-- snr=");
    Serial.print(lastSnr, 2);
  }

  // HR prints on its own segment regardless of BPM state,
  // but only when not in motion lockout and within hold window
  if (hrValid && !motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS)) {
    Serial.print(" HR:");
    Serial.print(lastHrBpm);
    Serial.print(" hr_snr=");
    Serial.print(lastHrSnr, 2);
  } else {
    hrValid = false;
  }

  Serial.println();

float displayDist = roundf(filtDistM * 10.0f) / 10.0f;  // show only 1 decimal
statusLcd.showVitals(lastHrBpm,
                     hrValid && !motionLockout && (now - lastHrGoodMs < CARDIAC_HOLD_MS),
                     max(11, lastBpm),
                     bpmValid && !movingNow,
                     filtDistM,
                     hasDistance,
                     detected);
}

// ====================== SETUP/LOOP ======================

void radar2Setup(bool initSerial, bool initLcd) {
  if (initSerial) {
    Serial.begin(115200);
```

```
    delay(1200);
  }

  Serial.println("HMMD breathing parser (overarching classifier + band filters)");
  if (initLcd && !statusLcd.begin()) {
    Serial.println("LCD1602 init failed: no PCF8574 found on 0x20-0x27 or 0x38-0x3F");
  } else if (initLcd) {
    Serial.print("LCD1602 found at 0x");
    Serial.println(statusLcd.address(), HEX);
  }

#if defined(ARDUINO_UNOR4_WIFI)
  RADAR_SERIAL_PORT.begin(115200);
  Serial.println("UNO R4 radar serial on D0 (RX) and D1 (TX) via Serial1");
#else
  RADAR_SERIAL_PORT.begin(115200, SERIAL_8N1, RADAR_RX_PIN, RADAR_TX_PIN);
#endif
  resetParser();

  for (int i = 0; i < SAMPLES; i++) {
    vReal[i] = 0.0;
    vImag[i] = 0.0;
    sampleRing[i] = 0.0;
  }

  radar2RecoverLink();
  delay(100);
}

void radar2LoopStep() {
  if (Serial.available()) {
    char c = Serial.read();
    if (c == 'r') {
        subjectDistLocked = false;
        lockedSubjectDistM = 0.0f;
        subjectDistDivergeMs = 0;
        clearSamples();
```

```
        lastBpm = -1;
        bpmValid = false;
        clsCount = 0;
        clsWrite = 0;
        classifierBand = BAND_UNKNOWN;
        bpmTrackerInit = false;
        Serial.println("Reset");
    }
}
    int budget = 16;  // or 16/64, tune as needed
  while (budget-- > 0 && RADAR_SERIAL_PORT.available()) {
    uint8_t b = (uint8_t)RADAR_SERIAL_PORT.read();
    radarByteCount++;
    parseRawByte(b);
}

   unsigned long now = millis();
    if ((now - lastGoodFrameMs > MODE_WATCHDOG_MS) &&
        (now - lastModeCmdMs > MODE_RETRY_GAP_MS)) {
      sendReportModeCmd();
      resetParser();
    }

      if ((now - lastRadarDiagMs) >= 1000UL) {
        unsigned long bytesPerSec = radarByteCount - lastRadarDiagByteCount;
        unsigned long framesPerSec = radarFrameCount - lastRadarDiagFrameCount;
        lastRadarDiagByteCount = radarByteCount;
        lastRadarDiagFrameCount = radarFrameCount;
        lastRadarDiagMs = now;
      }

    updateSampling();
    analyzeBreathingFast();
    analyzeBreathing();
    analyzeCardiac();
    static unsigned long lastDisplayMs = 0;
      if (now - lastDisplayMs >= 600) {         // 600 ms is perfect for human eyes
```

```
        printStatus();                  // keeps your existing Serial + LCD logic
        lastDisplayMs = now;
    }
}

#ifndef RADAR2_EMBEDDED
void setup() {
  radar2Setup(true, true);
}

void loop() {
  radar2LoopStep();
}
#endif
```

11. Waveshare Radar2Types header

```
#pragma once

// ====================== TIMING CONSTANTS ======================
#define CARDIAC_HOLD_MS  5000UL   // hold last valid HR reading this long

// ====================== BAND CONFIG ======================

struct BandCfg {
  const char* name;
  float fMinHz, fMaxHz;
  int bpmMin, bpmMax, restMax;
  float snrBase, snrHigh, peakMin;
};

struct BandResult {
  bool valid;
  int bpm;
  double peakMag, snr, score;
  int peakBin;
};
```

```
struct BandTrack {
  bool initialized;
  float filteredBpm, confidence;
};

enum BandIndex : uint8_t {
  BAND_ADULT = 0, BAND_CHILD = 1, BAND_INFANT = 2, BAND_UNKNOWN = 255
};
```

12. Waveshare Radar2Msg header

```
#pragma once
/*
 * Radar2VitalsMsg.h
 * ——————————————————
 * Defines the layout of the Radar2 vitals piggyback payload that rides in
 * spare bytes 14–19 of the UWB responder reply frame, plus the struct and
 * unpack helper used on the initiator side.
 *
 * Zero dependencies on uwb.h or the DW3000 stack.
 * Include it wherever you need to pack or unpack vitals data.
 */

#include <stdint.h>

// —— Frame byte offsets ——————————————————————————————————————————
// The UWB responder frame is 36 bytes. Bytes 0–13 are used by the ranging
// protocol (header, sequence number, function code, t_reply timestamp).
// Bytes 14–19 are spare and carry Radar2 vitals:
//
//   [14]     flags      bit0=bpmValid  bit1=hrValid  bit2=distValid
//   [15]     BPM        breathing rate  (uint8, 0–255)
//   [16]     HR         heart rate      (uint8, 0–255)
//   [17..19]  dist cm    filtDistM×100   uint24 little-endian
#define RADAR2_FLAGS_IDX   14
#define RADAR2_BPM_IDX     15
#define RADAR2_HR_IDX      16
#define RADAR2_DIST_IDX    17
```

```
#define RADAR2_FLAGS_BPM   0x01
#define RADAR2_FLAGS_HR    0x02
#define RADAR2_FLAGS_DIST  0x04

// ── Decoded vitals ──────────────────────────────────────────────────────────
struct Radar2Vitals {
    bool  active;    // true when at least one field is valid
    bool  bpmValid;
    bool  hrValid;
    bool  distValid;
    int   bpm;       // breathing rate BPM  (-1 = unavailable)
    int   hr;        // heart rate BPM      (-1 = unavailable)
    float distM;     // distance in metres  ( 0 = unavailable)
};

// ── Unpack helper (initiator side) ──────────────────────────────────────────
// Pass the raw received frame buffer. Fills `out` from bytes 14–19.
inline void unpackRadar2Vitals(const uint8_t* frameBuf, Radar2Vitals& out) {
    out = {false, false, false, false, -1, -1, 0.0f};
    if (!frameBuf) return;

    const uint8_t flags = frameBuf[RADAR2_FLAGS_IDX];

    if (flags & RADAR2_FLAGS_BPM) {
        out.bpmValid = true;
        out.bpm      = (int)frameBuf[RADAR2_BPM_IDX];
    }
    if (flags & RADAR2_FLAGS_HR) {
        out.hrValid = true;
        out.hr      = (int)frameBuf[RADAR2_HR_IDX];
    }
    if (flags & RADAR2_FLAGS_DIST) {
        out.distValid = true;
        const uint32_t cm = (uint32_t) frameBuf[RADAR2_DIST_IDX]
                          | ((uint32_t)frameBuf[RADAR2_DIST_IDX + 1] << 8)
                          | ((uint32_t)frameBuf[RADAR2_DIST_IDX + 2] << 16);
```

```
    out.distM = cm / 100.0f;
  }
  out.active = (flags != 0);
}
```

13. Waveshare Radar 1602 LCD screen

```
#pragma once

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

#ifndef RADAR_LCD_ADDR
#define RADAR_LCD_ADDR 0x27
#endif

#ifndef RADAR_LCD_COLS
#define RADAR_LCD_COLS 16
#endif

#ifndef RADAR_LCD_ROWS
#define RADAR_LCD_ROWS 2
#endif

class RadarLcd1602 {
public:
  explicit RadarLcd1602(uint8_t address = RADAR_LCD_ADDR) : address_(address) {}

  bool begin() {
    delay(120);

    if (!selectWorkingAddress()) {
      ready_ = false;
      return false;
    }

    ready_ = true;
    labelsDrawn_ = false;
```

```
  expanderWrite(LCD_BACKLIGHT);
  delay(20);

  write4(0x30, false);
  delay(10);
  write4(0x30, false);
  delay(5);
  write4(0x30, false);
  delay(5);
  write4(0x20, false);
  delay(5);

  command(0x28);
  command(0x06);
  command(0x0C);
  clear();
  drawLabelsIfNeeded();
  return true;
}

uint8_t address() const {
  return address_;
}

void clear() {
  if (!ready_) return;
  command(0x01);
  delay(2);
  labelsDrawn_ = false;
}

void setCursor(uint8_t col, uint8_t row) {
  if (!ready_) return;
  static const uint8_t rowOffsets[] = {0x00, 0x40, 0x14, 0x54};
  if (row >= RADAR_LCD_ROWS) row = RADAR_LCD_ROWS - 1;
  if (col >= RADAR_LCD_COLS) col = RADAR_LCD_COLS - 1;
```

```
  command(0x80 | (rowOffsets[row] + col));
}

void printFixed(uint8_t col, uint8_t row, const char *text) {
  if (!ready_) return;
  char line[RADAR_LCD_COLS + 1];
  memset(line, ' ', RADAR_LCD_COLS);
  line[RADAR_LCD_COLS] = '\0';
  if (text) {
    for (uint8_t i = 0; i < RADAR_LCD_COLS && text[i] != '\0'; i++) {
      line[i] = text[i];
    }
  }
  setCursor(col, row);
  for (uint8_t i = 0; i < RADAR_LCD_COLS; i++) {
    writeByte((uint8_t)line[i], true);
  }
}

void showVitals(int hrBpm, bool hrValid,
                int brBpm, bool brValid,
                float radarDistM, bool radarDistValid,
                bool detected) {
  char hrTxt[4];
  char brTxt[4];
  char radarTxt[8];
  char detTxt[4];

  drawLabelsIfNeeded();

  if (hrValid && hrBpm >= 0 && hrBpm <= 999) snprintf(hrTxt, sizeof(hrTxt), "%d", hrBpm);
  else snprintf(hrTxt, sizeof(hrTxt), "--");

  if (brValid && brBpm >= 0 && brBpm <= 999) snprintf(brTxt, sizeof(brTxt), "%d", brBpm);
  else snprintf(brTxt, sizeof(brTxt), "--");

  if (radarDistValid && isfinite(radarDistM)) snprintf(radarTxt, sizeof(radarTxt), "%.2fm", radarDistM);
```

```
    else snprintf(radarTxt, sizeof(radarTxt), "--");

    snprintf(detTxt, sizeof(detTxt), "%s", detected ? "Y" : "N");

    writeFieldIfChanged(3, 0, hrTxt, lastHrText_, sizeof(lastHrText_) - 1);
    writeFieldIfChanged(10, 0, brTxt, lastBrText_, sizeof(lastBrText_) - 1);
    writeFieldIfChanged(5, 1, radarTxt, lastRadarText_, sizeof(lastRadarText_) - 1);
    writeFieldIfChanged(15, 1, detTxt, lastDetText_, sizeof(lastDetText_) - 1);
  }

 private:
  static constexpr uint8_t LCD_RS = 0x01;
  static constexpr uint8_t LCD_RW = 0x02;
  static constexpr uint8_t LCD_E = 0x04;
  static constexpr uint8_t LCD_BACKLIGHT = 0x08;

  uint8_t address_;
  uint8_t backlightMask_ = LCD_BACKLIGHT;
  bool ready_ = false;
  bool labelsDrawn_ = false;
  char lastHrText_[4] = "";
  char lastBrText_[4] = "";
  char lastRadarText_[6] = "";
  char lastDetText_[4] = "";

  void command(uint8_t value) {
    writeByte(value, false);
    if (value == 0x01 || value == 0x02) delay(2);
    else delayMicroseconds(100);
  }

  void writeByte(uint8_t value, bool isData) {
    write4(value & 0xF0, isData);
    write4((value << 4) & 0xF0, isData);
  }

  void write4(uint8_t nibble, bool isData) {
```

```
  uint8_t value = (nibble & 0xF0) | backlightMask_ | (isData ? LCD_RS : 0);
  value &= ~LCD_RW;
  expanderWrite(value);
  expanderWrite(value | LCD_E);
  delayMicroseconds(1);
  expanderWrite(value & ~LCD_E);
  delayMicroseconds(50);
}

void expanderWrite(uint8_t value) {
  Wire.beginTransmission(address_);
  Wire.write(value);
  Wire.endTransmission();
}

bool probeAddress(uint8_t address) {
  Wire.beginTransmission(address);
  return Wire.endTransmission() == 0;
}

bool selectWorkingAddress() {
  if (probeAddress(address_)) return true;

  for (uint8_t addr = 0x20; addr <= 0x27; addr++) {
    if (probeAddress(addr)) {
      address_ = addr;
      return true;
    }
  }

  for (uint8_t addr = 0x38; addr <= 0x3F; addr++) {
    if (probeAddress(addr)) {
      address_ = addr;
      return true;
    }
  }
```

```
    return false;
  }

  void drawLabelsIfNeeded() {
    if (!ready_ || labelsDrawn_) return;
    printFixed(0, 0, "HR:    BR:      ");
    printFixed(0, 1, "Dist:      Det:     ");
    lastHrText_[0] = '\0';
    lastBrText_[0] = '\0';
    lastRadarText_[0] = '\0';
    labelsDrawn_ = true;
  }

  void writeField(uint8_t col, uint8_t row, const char *text, uint8_t width) {
    if (!ready_) return;
    setCursor(col, row);
    for (uint8_t i = 0; i < width; i++) {
      char ch = (text && text[i] != '\0') ? text[i] : ' ';
      writeByte((uint8_t)ch, true);
    }
  }

  void writeFieldIfChanged(uint8_t col, uint8_t row, const char *text, char *cachedText, uint8_t width) {
    char nextText[8];
    for (uint8_t i = 0; i < width; i++) {
      nextText[i] = (text && text[i] != '\0') ? text[i] : ' ';
    }
    nextText[width] = '\0';

    if (strncmp(nextText, cachedText, width) == 0) return;

    writeField(col, row, nextText, width);
    memcpy(cachedText, nextText, width + 1);
  }
};
```

13. Anchor/Responder UWB.cpp header

```
#define MAIN_U2
#include "uwb.h"
#include "dw3000.h"

dwt_config_t config = {
    5,
    DWT_PLEN_128,
    DWT_PAC8,
    9,
    9,
    1,
    DWT_BR_6M8,
    DWT_PHRMODE_STD,
    DWT_PHRRATE_STD,
    (129 + 8 - 8),
    DWT_STS_MODE_OFF,
    DWT_STS_LEN_64,
    DWT_PDOA_M0
};

extern dwt_txconfig_t txconfig_options;

uint8_t tx_msg[] = {0x41, 0x88, 0, 0xCA, 0xDE, 0, 0, UID, 0, FUNC_CODE_INTER,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0};

uint8_t rx_msg[] = {0x41, 0x88, 0, 0xCA, 0xDE, 0, 0, UID, 0, FUNC_CODE_INTER,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0};

uint8_t frame_seq_nb = 0;
uint8_t rx_buffer[NUM_NODES - 1][BUF_LEN];
uint32_t status_reg = 0;
```

```
int counter = 0;
int ret = 0;
uint64_t rx_ts[NUM_NODES - 1];
uint64_t desired_tx_ts;
uint64_t real_tx_ts;
uint64_t tx_time;
uint64_t t_reply;
uint64_t t_round;
double tof = 0.0;
double distance = 0.0;
unsigned long previous_debug_millis = 0;
unsigned long current_debug_millis = 0;
int target_uids[NUM_NODES - 1];
const char* uwb_last_error = "UWB:idle";

static void set_target_uids() {
#ifdef MAIN_U1
    target_uids[0] = U2;
#elif defined(MAIN_U2)
    target_uids[0] = U1;
#endif
}

bool start_uwb() {
    int idleRetries = 10;
    while (!dwt_checkidlerc()) {
        if (--idleRetries <= 0) {
            uwb_last_error = "UWB:idle rc";
            Serial.println("UWB stage: idle_rc timeout");
            return false;
        }
        delay(10);
    }

    if (dwt_initialise(DWT_DW_INIT) == DWT_ERROR) {
        if (dwt_initialise(DWT_DW_IDLE_RC) == DWT_ERROR) {
            uwb_last_error = "UWB:init hw";
```

```
            Serial.println("UWB stage: dwt_initialise failed");
            return false;
        }
    }

    dwt_setxtaltrim(0x2E);
    delay(5);
    dwt_setleds(DWT_LEDS_DISABLE);

    if (dwt_configure(&config)) {
        uwb_last_error = "UWB:cfg";
        Serial.println("UWB stage: dwt_configure failed");
        return false;
    }

    dwt_configuretxrf(&txconfig_options);
    dwt_setrxantennadelay(RX_ANT_DLY);
    dwt_settxantennadelay(TX_ANT_DLY);
    dwt_setrxaftertxdelay(TX_TO_RX_DLY_UUS);
    dwt_setrxtimeout(0);
    dwt_setlnapamode(DWT_LNA_ENABLE | DWT_PA_ENABLE);

    set_target_uids();
    uwb_last_error = "UWB:ready";
    Serial.println("UWB stage: ready");
    return true;
}

bool responder() {
    if (counter == 0) {
        // Arm RX and immediately return — poll next iteration
        dwt_rxenable(DWT_START_RX_IMMEDIATE);
        counter = -1;
        return false;
    }

    // Non-blocking: check status once and return if nothing ready yet
```

```
status_reg = dwt_read32bitreg(SYS_STATUS_ID);
if (!(status_reg & (SYS_STATUS_RXFCG_BIT_MASK | SYS_STATUS_ALL_RX_ERR))) {
    return false;
}

// Reset counter so frame-processing logic uses index 0
counter = 0;

if (status_reg & SYS_STATUS_RXFCG_BIT_MASK) {
    dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_RXFCG_BIT_MASK);
    dwt_readrxdata(rx_buffer[counter], BUF_LEN, 0);
    rx_ts[counter] = get_rx_timestamp_u64();

    if (rx_buffer[counter][MSG_FUNC_IDX] == FUNC_CODE_RESET) {
        counter = 0;
        return false;
    }

    if (rx_buffer[counter][MSG_SID_IDX] != target_uids[counter]) {
        dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_ALL_RX_TO | SYS_STATUS_ALL_RX_ERR);
        counter = 0;
        return false;
    }

    ++counter;

} else {
    // Timeout — mirrors initiator reset path
    dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_ALL_RX_TO | SYS_STATUS_ALL_RX_ERR);
    counter = 0;
    ++frame_seq_nb;
    return false;
}

if (counter == NUM_NODES - 1) {
    // Build and send delayed response — this is the responder's "compute" step
    tx_time = (rx_ts[0] + (RX_TO_TX_DLY_UUS * UUS_TO_DWT_TIME)) >> 8;
```

```
        desired_tx_ts = (((uint64_t)(tx_time & 0xFFFFFFFEUL)) << 8) + TX_ANT_DLY;
        dwt_setdelayedtrxtime(tx_time);

        uint64_t t_reply_duration = desired_tx_ts - rx_ts[0];
        uint32_t t_reply_packed = (uint32_t)(t_reply_duration & 0xFFFFFFFFUL);
        tx_msg[10] = (t_reply_packed >>  0) & 0xFF;
        tx_msg[11] = (t_reply_packed >>  8) & 0xFF;
        tx_msg[12] = (t_reply_packed >> 16) & 0xFF;
        tx_msg[13] = (t_reply_packed >> 24) & 0xFF;

        tx_msg[MSG_SN_IDX] = frame_seq_nb;
        dwt_writetxdata((uint16_t)(MSG_LEN), tx_msg, 0);
        dwt_writetxfctrl((uint16_t)(MSG_LEN), 0, 1);
        ret = dwt_starttx(DWT_START_TX_DELAYED);

        if (ret == DWT_SUCCESS) {
            while (!(dwt_read32bitreg(SYS_STATUS_ID) & SYS_STATUS_TXFRS_BIT_MASK)) {}
            dwt_write32bitreg(SYS_STATUS_ID, SYS_STATUS_TXFRS_BIT_MASK);
        }

        counter = 0;
        ++frame_seq_nb;
        return true;
    }

    return false;
}
```

Appendix D – AI Prompts

[1a] Prompt: “What are the most effective ways to optimize power consumption in a dual-rail Li-ion–powered radar sensing system?”

[1b] ChatGPT Outcome: To optimize power consumption in a dual-rail Li-ion–powered radar sensing system, it’s important to consider both hardware efficiency and firmware-level power management. On the hardware side, using synchronous buck-boost converters with low quiescent current (IQ < 50 µA)

significantly reduces idle losses. Separate regulation for the 3.3 V and 5 V rails ensures that conversion efficiency is maintained across load variations. Additionally, implementing load switches or MOSFET-based power gating on subsystems like the display, radar, and communication modules allows selective power delivery only when necessary.

On the firmware side, duty-cycling the radar—activating it only during scan intervals—can reduce average current by more than 60%. The ESP32's deep-sleep mode should be leveraged between radar bursts, and periodic wake-ups scheduled via timers or interrupts. Adjusting radar sampling frequency dynamically based on signal activity can further improve efficiency. Real-time current monitoring via a shunt-based sensor (e.g., INA219) can help profile load behavior and validate design assumptions. Together, these techniques establish a layered strategy that improves both runtime and thermal performance without compromising system responsiveness.

[1c] Claude Outcome: Optimizing power consumption in a dual-rail Li-ion-powered radar sensing system requires focusing on a few key areas. The single most impactful strategy is maximizing sleep time between radar measurements, since the RF transmitter typically dominates power consumption during active operation. Use intelligent duty cycling with motion sensors or other low-power triggers to activate the radar only when needed, and implement deep sleep modes that draw only microamperes when idle.

On the hardware side, use high-efficiency DC-DC converters (>90% efficiency) with low quiescent current for both power rails, and consider power gating to completely disconnect unused sections between measurements. Dynamic voltage scaling can adjust rail voltages based on real-time performance requirements. Select radar transceivers and microcontrollers specifically designed for low-power operation with fast sleep-to-active transitions, and use hardware accelerators for signal processing tasks like FFTs rather than relying on general-purpose processors.

For radar operation itself, optimize chirp parameters to balance performance with energy per measurement, reduce sampling rates when no targets are detected, and implement adaptive frame rates based on scene activity. Accurate battery monitoring through fuel gauge ICs enables intelligent power management decisions throughout the system's operational life. With proper optimization across these areas, you can achieve 100-1000x power reduction compared to continuous operation, dramatically extending battery life.

[2a] Prompt: “What filtering techniques can be applied to reduce false positives in a short-range radar-based life-detection system?”

[2b] ChatGPT Outcome: In short-range radar-based life-detection systems, false positives often arise from motion artifacts, environmental reflections, and electrical noise. Applying digital signal filtering at both the hardware and software

stages is crucial. A Butterworth low-pass filter effectively removes high-frequency noise while maintaining signal shape, whereas Chebyshev Type II filters provide steeper roll-off for environments with dense interference. For dynamic background suppression, adaptive Kalman filtering can continuously update prediction models based on signal variance, improving accuracy over time.

Additionally, moving-average filters help stabilize respiration or heartbeat signals, particularly when integrated with real-time peak detection algorithms. When computational resources allow, a combination of FFT-based noise analysisand band-pass filtering centered on the human respiration frequency range (0.1–0.4 Hz) yields optimal discrimination between genuine motion and background noise. These methods can be implemented in MATLAB or directly within the radar's embedded DSP pipeline.

[2c] Claude Outcome: Reducing false positives in a short-range radar-based life-detection system requires combining multiple filtering techniques across the signal processing chain. Start with robust clutter removal using high-pass filtering or background subtraction to eliminate static objects, which are often the primary source of false alarms. Implement adaptive clutter cancellation that continuously updates the background model to handle slowly changing environments like swaying vegetation or temperature-related drift.

For vital sign detection, apply narrow bandpass filtering around expected heartbeat (0.8-2 Hz) and respiration (0.1-0.5 Hz) frequencies to isolate physiological signatures while rejecting out-of-band noise. Use phase-based processing techniques rather than just amplitude, as phase information provides more reliable vital sign extraction and is less susceptible to multipath interference. Implement range gating to focus on specific distance bins where a person is likely located, which dramatically reduces the search space and false alarm potential. Apply temporal consistency checks by requiring life signs to persist for multiple consecutive frames before declaring a detection, filtering out transient disturbances like passing vehicles or moving objects. Use machine learning classifiers trained on real vital sign data versus common false alarm sources to distinguish genuine human signatures from mimics like fans, machinery vibration, or animal movement. Cross-validation techniques like requiring simultaneous detection of both heartbeat and respiration patterns, with physiologically plausible phase relationships between them, significantly improve specificity. Finally, implement adaptive thresholding that adjusts detection sensitivity based on environmental noise levels and clutter statistics, preventing both missed detections in challenging conditions and excessive false alarms in benign environments.

[3a] Prompt: "What communication protocol offers the best balance between speed, reliability, and wiring simplicity for interfacing a radar baseboard with an ESP32 microcontroller?"

[3b] ChatGPT Outcome: For interfacing a radar baseboard with an ESP32

microcontroller, SPI (Serial Peripheral Interface) provides the optimal balance of speed, reliability, and design simplicity. SPI offers data rates exceeding 10 Mbps, ideal for transferring radar frame data with minimal latency. Its full-duplex communication mode and support for multiple slaves make it suitable for multi-device setups involving a display or secondary sensors.

ChatGPT also discussed signal integrity considerations, recommending short traces and proper grounding to minimize noise coupling. In systems requiring asynchronous logging or debug data, UART serves as a complementary channel. The model also highlighted that SPI's lower overhead and deterministic timing outperform I²C for time-critical radar data streams, making it the preferred protocol for the main data path.

[3c] Claude Outcome: For interfacing a radar baseboard with an ESP32 microcontroller, SPI (Serial Peripheral Interface) typically offers the best balance between speed, reliability, and wiring simplicity. SPI provides high data rates of 10-80 MHz on the ESP32, which is more than adequate for transferring radar data frames, and requires only four wires (MOSI, MISO, SCLK, and CS). The protocol is straightforward to implement with well-supported libraries, minimal CPU overhead through DMA support, and deterministic timing that's crucial for real-time radar data acquisition.

For most radar applications requiring raw or minimally processed data transfer, SPI's combination of high throughput, low latency, hardware acceleration support on the ESP32, and relatively simple four-wire connection makes it the optimal choice. The ESP32 has multiple SPI peripherals available, allowing you to dedicate one exclusively to radar communication while using others for displays or SD cards if needed.

[4a] Prompt: "Which enclosure materials are best suited for housing a radar-based life-detection device that must minimize interference and remain lightweight?"

[4b] ChatGPT Outcome: For this type of radar-based system, I recommend using polycarbonate-ABS (PC-ABS) due to its excellent mechanical durability, low dielectric constant, and compatibility with RF applications. PC-ABS maintains radar signal transparency and provides sufficient impact protection for field use. I would avoid metallic or carbon-filled enclosures since they can reflect or absorb radar energy, reducing signal quality.

To improve environmental performance, I suggest an IP54 or higher enclosure rating, achieved by using silicone gaskets and minimal seam lines. A conformal coating on internal PCBs further protects against humidity and dust. Thermal management can be achieved with discreet vents or internal heat spreaders while maintaining RF clarity. Overall, PC-ABS offers the best balance between strength, low weight, and radar transparency.

[4c] Perplexity Outcome: For housing a radar-based life-detection device, the best enclosure materials balance RF transparency, mechanical protection, and weight. Plastics are generally the optimal choice, with ABS, polycarbonate, and PETG offering excellent radar transparency across common frequencies (24 GHz, 60 GHz, 77 GHz bands) while providing good impact resistance and being lightweight. These materials introduce minimal signal attenuation, typically less than 1-2 dB, which preserves both transmit power and receive sensitivity critical for life detection.

Composite materials like fiberglass or carbon fiber offer excellent strength-to-weight ratios, but carbon fiber is electrically conductive and will block radar signals, so it should only be used in non-critical areas away from the antenna aperture. If you need EMI shielding for the electronics section to prevent interference with the radar receiver, use internal metal partitions or conductive coatings in compartments separate from the antenna, creating isolated RF zones. For outdoor applications, ensure whatever material you choose has good UV stability and moisture resistance, as environmental degradation can change dielectric properties over time and affect radar performance.

[5a] Prompt: “What are effective methods for visualizing radar-based motion and breathing data on a 4.3-inch TFT display?”

[5b] ChatGPT Outcome: For visualizing radar data in real time, I recommend using waveform and bar graph visualizations to represent both motion amplitude and respiration rate. A scrolling waveform offers a clear view of breathing patterns, while a color-coded bar indicator can show motion intensity. I suggest implementing double-buffered rendering and partial screen updates to prevent flicker or tearing, especially when displaying continuous data streams.

In addition, I would apply adaptive brightness control based on ambient light, along with high-contrast color schemes for readability in bright conditions. Icons or status indicators could display connection state, detection status, or error messages. Overall, the goal is to keep the display simple, responsive, and visually intuitive for users operating in high-stress or outdoor environments.

[5c] Claude Outcome: For visualizing radar-based motion and breathing data on a 4.3-inch TFT display, focus on real-time waveforms and simple visual indicators that convey information at a glance. Display breathing and heartbeat as scrolling time-domain waveforms, similar to medical monitors, with the past few seconds of data visible so users can quickly assess regularity and presence of vital signs. Use color coding to indicate signal quality or detection confidence, such as green for strong signals, yellow for marginal, and red for no detection or poor quality.

For motion detection, use a simple zone-based display showing which range bins or spatial regions contain detected movement, represented as a top-down or

side-view map with color intensity indicating motion magnitude. A waterfall or spectrogram display can effectively show frequency content over time, useful for debugging and detailed analysis, though this may be too technical for end users. Keep the interface uncluttered given the small screen size—prioritize the most actionable information and use tabbed views or button presses to access detailed diagnostics. Finally, ensure your refresh rate matches your radar update rate but limit unnecessary screen redraws to minimize flicker and power consumption, particularly important if battery-powered.

[6a] I am conducting research on INS modules with GNSS. We are looking into the M135 module, can you provide addition details and more info on others for comparison?

[6b] **u-blox ZED-F9R-03B**
Key links:

- Product page (u-blox):
  - https://www.u-blox.com/en/product/zed-f9r-module u-blox
- Datasheet (PDF):
  - https://content.u-blox.com/sites/default/files/documents/ZED-F9R-03B_DataSheet_UBX-22024085.pdfu-blox
- Distributor examples (with datasheet link):
  - Mouser: https://www.mouser.com/ProductDetail/u-blox/ZED-F9R-03B Mouser Electronics
  - Digi-Key: https://www.digikey.com/en/products/detail/u-blox/ZED-F9R-03B/18880544 DigiKey

- Dual-band, multi-constellation **high-precision GNSS + 3D IMU** module with on-board sensor fusion for dead-reckoning. u-blox+1
- Supports RTK / PPP-RTK and real-time position update rates up to **30 Hz**. u-blox+1
- Supply: **2.7–3.6 V**, typical operation power in the mid-hundreds of mW range (matches your ~0.255 W line). u-blox+1
- Package size: **17 × 22 × 2.4 mm** LGA module. u-blox+1

u-blox NEO-M9V-20B

Key links:

- Product page (u-blox):
  - https://www.u-blox.com/en/product/neo-m9v-module u-blox
- Datasheet (PDF):
  - https://content.u-blox.com/sites/default/files/NEO-M9V-20B_DataSheet_UBX-21029781.pdf (or equivalent mirror) u-blox+2Javan Electronic+2
- Distributor examples:

    - Mouser: https://www.mouser.com/ProductDetail/u-blox/NEO-M9V-20B Mouser Electronics
    - Richardson RFPD: https://shop.richardsonrfpd.com/Products/Product/NEO-M9V-20B RichardsonRFPD

- L1 multi-constellation GNSS receiver with **onboard 6-axis IMU** and **untethered / automotive dead-reckoning (UDR/ADR)** capability. u-blox+1
- Based on the u-blox M9 platform; supports GPS, Galileo, GLONASS, BeiDou. u-blox+1
- Typical update rates up to **50 Hz** for dead-reckoning, with default 1 Hz GNSS. u-blox
- Supply: **2.7–3.6 V**, module size **16 × 12.2 × 2.4 mm** (matches your table). u-blox+2RichardsonRFPD+2

ST Teseo-VIC3DA

Key links:

- ST official product page:
    - https://www.st.com/content/st_com/en/campaigns/teseo-vic3da-gnss-dead-reckoning-imu.htmlSTMicroelectronics
- Datasheet (PDF):
    - https://www.st.com/resource/en/datasheet/teseo-vic3da.pdf STMicroelectronics
- Distributor examples:
    - Mouser: https://www.mouser.com/ProductDetail/STMicroelectronics/TESEO-VIC3DA Mouser Electronics
    - Future Electronics: https://www.futureelectronics.com/npi/stmicroelectronics-teseo-vic3da_automotive-gnss-dead-reckoning-module Future Electronics

- Automotive **GNSS dead-reckoning module** with integrated **6-axis IMU**, based on the TeseoIII GNSS receiver. STMicroelectronics+2STMicroelectronics+2
- Supports multi-constellation (GPS, Galileo, GLONASS, BeiDou, QZSS) with embedded dead-reckoning firmware. STMicroelectronics+2STMicroelectronics+2
- Automotive-grade, AEC-Q100 qualified; includes assisted-GNSS features (A-GNSS, predictive, real-time assisted modes). STMicroelectronics+1
- Supply: **~3.0–3.3 V**, typical current ~59 mA (≈0.161–0.19 W depending on conditions); package around **16 × 12.2 × 2.4 mm**. STMicroelectronics+2Elfa+2

M5Stack M135 – GNSS Module (NEO-M9N + Sensors)

Key links:

- M5Stack docs page (GNSS Module / M135):
  - https://docs.m5stack.com/en/module/GNSS%20Module M5Stack Docs
- M5Stack shop product page:
  - https://shop.m5stack.com/products/gnss-module-with-barometric-pressure-imu-magnetometer-sensorsm5stack-store
- Datasheet / module PDF:
  - https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/docs/static/pdf/static/en/module/GNSS%20Module.pdf M5Stack
- Unofficial detail page (lists sensor part numbers):
  - https://m5stack.lang-ship.com/catalog/products/module/m135_gnss-module/ M5Stack Unofficial Guide

- GNSS "sensor carrier" module built around **u-blox NEO-M9N** plus:
  - Bosch **BMI270** 6-axis IMU
  - Bosch **BMM150** 3-axis magnetometer
  - Bosch **BMP280** barometric pressure sensor M5Stack+2M5Stack Docs+2
- Supports four-constellation L1 GNSS reception (GPS, GLONASS, Galileo, BeiDou), up to **25 Hz** navigation update. M5Stack+1
- Interfaces via M5BUS (UART, PPS, configurable pins) for use with M5Stack/ESP32 cores; not a full INS by itself – **fusion is done on the host MCU**. M5Stack+1
- Typical supply **3.0–3.3 V**, multi-sensor integration packaged in an M5Stack module shell (**~54 × 54 × 13.1 mm**).

[7a] I am comparing barometers for a handheld device detecting humans. Can you provide some to compare?

[7b] Bosch BMP390

- **Type / Sensor Suite**: Barometric Pressure + Temperature Sensor
- **Key Features**:
  - Digital absolute pressure sensor (300 hPa to 1250 hPa) with typical relative accuracy ±3 Pa (~±0.25 m altitude) in 700–1100 hPa, 25–40 °C. Bosch Sensortec+1
  - Ultra-low power: typical 3.2 µA at 1 Hz update rate. Mouser Electronics+1
  - Interface: I²C up to 3.4 MHz, SPI up to 10 MHz. Mouser Electronics+1

    - Supply voltage: VDD 1.65 V–3.6 V; VDDIO 1.2 V–3.6 V. Mouser Electronics+1
    - Package size: 2.0 × 2.0 × 0.75 mm³ LGA. Bosch Sensortec+1
- **Good for INS/altitude fusion because**: Very high precision and low power make it ideal for altitude changes and we can fuse with GNSS/IMU for better vertical tracking.
- **Key Links (for your references section)**:
    - Datasheet PDF: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp390-ds002.pdf Bosch Sensortec
    - Product page: https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp390/ Bosch Sensortec
    - Distributor listing: https://www.mouser.com/ProductDetail/Bosch-Sensortec/BMP390 Mouser Electronics

Infineon DPS310

- **Type / Sensor Suite**: Barometric Pressure + Temperature Sensor
- **Key Features**:
    - High-precision digital barometric air pressure sensor; able to detect small altitude changes (~±0.02 m) in high precision mode. Adafruit Learn+1
    - Supply: typical ~3.3 V (in many boards) and interface via I²C or SPI. Adafruit Learn
    - Excellent resolution and low drift, targeted at mobile/portable/wearable devices. Octopart+1
- **Good for INS/altitude fusion because**: Enables fine vertical motion detection (important e.g., flights, drones) where altitude changes matter.
- **Key Links**:
    - Datasheet PDF: https://www.alldatasheet.com/datasheet-pdf/pdf/876653/INFINEON/DPS310.htmlAllDataSheet
    - Info page: https://www.infineon.com/products/sensor/pressure-sensors/pressure-sensors-for-iot Infineon

Bosch BMP280

- **Type / Sensor Suite**: Barometric Pressure + Temperature Sensor
- **Key Features**:
    - Pressure sensing range about 300–1100 hPa, absolute accuracy ~±1 hPa (≈±1 m altitude) under typical conditions. Bosch Sensortec+1
    - Very low power: e.g., ~2.7 µA at 1 Hz measurement. Bosch Sensortec+1
    - Compact package: 2.0 × 2.5 mm², height ~0.95 mm. Bosch Sensortec

    - Supply voltage: 1.71 V or above (depending) for the module. Bosch Sensortec+1
- **Good for INS/altitude fusion because**: Cost-effective baseline sensor — decent accuracy and widely used; good reference for comparison.
- **Key Links**:
    - Datasheet PDF: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001.pdf Bosch Sensortec
    - Product flyer: https://www.bosch-sensortec.com/media/boschsensortec/downloads/product_flyer/bst-bmp280-fl000.pdf Bosch Sensortec

ST LPS22HB

- **Type / Sensor Suite**: Barometric Pressure Sensor (Digital Output)
- **Key Features**:
    - Pressure range: 260 to 1260 hPa. STMicroelectronics+1
    - Supply voltage: 1.7 V to 3.6 V. STMicroelectronics
    - Output data rate (ODR) from 1 Hz up to 75 Hz. STMicroelectronics
    - Embedded FIFO, interrupt functions, I²C/SPI interface. STMicroelectronics
    - Package: ultra-compact, high shock survivability (22,000 g) for rugged applications. STMicroelectronics
- **Good for INS/altitude fusion because**: Good option when you need higher sample rate and compact rugged sensor for embedded navigation/altitude tasks.
- **Key Links**:
    - Datasheet PDF: https://www.st.com/resource/en/datasheet/lps22hb.pdf STMicroelectronics
    - Product page: https://www.st.com/en/mems-and-sensors/lps22hb.html STMicroelectronics

[8a] I am comparing magnetometers for a handheld device detecting humans. Can you provide some to compare?

[8b] Memsic MMC5983MA

- **Type / Sensor Suite**: 3-axis AMR magnetometer (digital)
- **Key Features**:
    - ±8 G full-scale, **18-bit** resolution, **~0.4 mG RMS noise**, **ODR up to 1000 Hz**; on-chip **SET/RESET (degauss)** to remove residual magnetization. memsic.com
    - **I²C (400 kHz)** and **SPI (10 MHz)** digital interfaces. memsic.com
    - Supply: **2.8–3.5/3.6 V** (VDD), low-profile **3.0 × 3.0 × 1.0 mm** LGA. DigiKey+1

- **Good for INS/heading fusion because**: Very low noise + high resolution and fast ODR help stabilize heading estimates and tolerate brief magnetic disturbances; SET/RESET improves repeatability. memsic.com
- **Key Links:**
  - Datasheet (PDF): MMC5983MA Magnetic Sensor. memsic.com
  - Product page (features, ODR, package): MMC5983MA. memsic.com
  - Distributor specs: Digi-Key / Mouser. DigiKey+1

ST LIS3MDL

- **Type / Sensor Suite**: 3-axis magnetometer (digital)
- **Key Features**:
  - **Selectable full scales**: ±4/±8/±12/±16 gauss; **16-bit** output; **ultra-low-power** operation modes. STMicroelectronics
  - **I²C (Std/Fast)** and **SPI** interfaces; **interrupt** and **self-test** features. STMicroelectronics+1
  - Supply: **1.9–3.6 V** (with **1.8 V IO** option). STMicroelectronics
- **Good for INS/heading fusion because**: Mature, widely used part with flexible ranges and low power—easy to integrate with common IMUs in 9-DoF stacks. STMicroelectronics+1
- **Key Links:**
  - Datasheet (PDF): LIS3MDL Digital Output Magnetic Sensor. STMicroelectronics
  - Product page: ST LIS3MDL. STMicroelectronics
  - App/guide (use & integration): Adafruit LIS3MDL guide. Adafruit Learn

QST QMC5883L

- **Type / Sensor Suite**: 3-axis magnetometer (digital), HMC5883L-class replacement
- **Key Features**:
  - 3-axis magnetic sensor with on-chip signal-conditioning **ASIC**; **I²C** digital output; small **LGA-16 (≈3 × 3 × 0.9 mm)** package. qstcorp.com+2qstcorp.com+2
  - Targeted for **compassing/navigation** in drones, robots, and handhelds. qstcorp.com
- **Good for INS/heading fusion because**: Low-cost, widely available alternative for basic e-compass/heading tasks when top-tier noise/linearity aren't critical. qstcorp.com
- **Key Links:**
  - Datasheet (PDF): QMC5883L 3-Axis Magnetic Sensor. qstcorp.com
  - Product page: QST QMC5883L. qstcorp.com

Bosch BMM150

- **Type / Sensor Suite**: 3-axis geomagnetic sensor (digital)
- **Key Features**:
  - **Ultra-low-voltage** operation: **VDD 1.62–3.6 V**, **VDDIO 1.2–3.6 V**; programmable performance/power modes; **interrupt engine**. Bosch Sensortec
  - Digital interfaces (**I²C/SPI**); used widely in phones/IMUs (note: device status is moving to **obsolete** at some distributors). Bosch Sensortec+1
  - Tiny WLCSP options (e.g., **~1.56 × 1.56 mm**); also found on many combo sensor boards. DigiKey
- **Good for INS/heading fusion because**: Proven workhorse magnetometer that pairs well with Bosch IMUs; many open drivers and examples available. Bosch Sensortec+1
- **Key Links:**
  - Datasheet (PDF): BMM150 Geomagnetic Sensor. Bosch Sensortec
  - Bosch Sensor API (reference driver): BMM150_SensorAPI (GitHub). GitHub
  - Distributor status example (obsolescence note, pkg): Digi-Key. DigiKey

[9a] I am doing research for a handheld device for detecting humans in emergency situations. I want to investigate ultra-wideband sensors. Can you provide some to compare?

[9b] Qorvo DWM3000EVB

- **Type / What it is**: Arduino-form-factor **evaluation board** for the DWM3000 UWB module (DW3110 inside).
- **Key features**:
  - Runs **IEEE 802.15.4z BPRF** (backward compatible with 802.15.4a); supports **Channel 5 (6.5 GHz)** & **Channel 9 (8.0 GHz)**. Symmetry Electronics+1
  - Works as a shield with many MCUs (Arduino header), flexible host selection (ST, Nordic, etc.). Symmetry Electronics
  - Pairs with DW3xxx/QM3xxx SDK (ranging, TDoA/PDoA examples; Nearby-Interaction integration in recent SDKs). Qorvo
- **Good for your project because**: fastest path to bring-up DW3110-class UWB on ESP32/STM32/Nordic dev boards; reference HW for antenna, power, IO.
- **Key links (for references)**:
  - Product page: Qorvo **DWM3000EVB**. Qorvo
  - Product brief (PDF): **DWM3000EVB Product Brief**. Symmetry Electronics
  - Quick start / guide (mirror): manuals.plus entry. Manuals+

ULM3-STM32 + DWM3000 (HaoruTech ULM3)

- **Type / What it is**: **UWB positioning module** that integrates **DWM3000** + **STM32F103** MCU + small OLED; can act as tag/anchor.
- **Key features**:
  - Based on **DW3000-series** (DWM3000 core), supports ranging/indoor positioning; firmware, PC tool & Android app provided. Haoru Tech+1
  - Interfaces via AT-style or open firmware; designed for quick evaluations, classroom demos, and anchor/tag networks. Haoru Tech
- **Good for your project because**: turn-key demo node when you want **on-board MCU** + UI to prototype quickly without wiring a separate host board.
- **Key links (for references)**:
  - User Manual (PDF): **ULM3_UserManual-EN**. Haoru Tech
  - Product listing (overview/features): RobotShop ULM3 page. RobotShop USA

Qorvo DWM1000 (DW1000-based module)

- **Type / What it is**: Legacy Decawave **UWB transceiver module** (pre-DW3000 generation) with integrated antenna/RF.
- **Key features**:
  - **IEEE 802.15.4-2011 UWB** compliant; **data rates 110 kb/s, 850 kb/s, 6.8 Mb/s**; supports multiple RF bands **3.5–6.5 GHz**. Qorvo
  - Designed to comply with **FCC & ETSI UWB masks**; SPI host interface; simple 2-way ranging or TDoA systems, **~10 cm** precision typical. Qorvo+1
- **Good for your project because**: widely used in earlier UWB designs; tons of community examples—useful **baseline** or drop-in for simple ranging.
- **Key links (for references)**:
  - Product page: **DWM1000** (Qorvo). Qorvo
  - Datasheet (PDF mirror): **DWM1000 Datasheet**. 91Chip
  - Product brief (PDF): **dwm1000-product-brief**. laskakit.cz

Qorvo DWM3000 (DW3110-based module)

- **Type / What it is**: Current-gen **UWB transceiver module** integrating antenna, RF, PMU, clock; **DW3110** inside.
- **Key features**:
  - **IEEE 802.15.4a / 802.15.4z BPRF** compliant; **Channel 5 & 9** operation; **~10 cm** location precision class. Qorvo
  - Supports **Two-Way Ranging**, **TDoA**, **PDoA** frameworks; FiRa-class ecosystem; DW3110 interoperable with **Apple U1/U2** devices (Nearby Interaction). Qorvo

    - SPI host interface; docs & SDKs available; integrates **always-on memory** and low-power modes (see datasheet). MikroElektronika Downloads
- **Good for your project because**: preferred module for **modern UWB** (FiRa/NI) with better coexistence and ecosystem support than DW1000-era parts.
- **Key links (for references)**:
    - Product page: **DWM3000** (Qorvo). Qorvo
    - Product brief (PDF): **DWM3000 Product Brief**. Mouser Electronics+1
    - DW3110 transceiver page (capabilities): **DW3110**. Qorvo

[10a] We are thinking about incorporating a geophone for a project detecting people. Can you provide me with some options to compare?

[10b] SM-24 Geophone (ION Geophysical)

- **Type / Sensor Suite**: Vertical component analog **geophone sensor** (velocity transducer).
- **Key Features**:
    - **Natural frequency:** 10 Hz ±5%
    - **Coil resistance:** ~375 Ω ± 5%
    - **Open circuit sensitivity:** 28.8 V/m/s ± 5%
    - **Distortion:** < 0.2% at 0.7 in/s p-p
    - **Damping:** 0.6 (nominal)
    - **Operating temperature:** −40 °C to +100 °C
    - **Typical housing:** 25.4 mm diameter, 36 mm height
- **Good for field use because**: Standard **industry-grade seismic element** used globally; high reliability and ruggedness make it ideal for both laboratory and outdoor vibration or impact detection systems.
- **Key Links**:
    - Datasheet (PDF): ION SM-24 Geophone Specification Sheet
    - Manufacturer overview: ION Geophysical SM-Series Sensors
    - Common distributor: GeoSpace Technologies SM-24

SM-4 Horizontal Geophone (Sercel)

- **Type / Sensor Suite**: Dual-axis **horizontal** geophone, primarily for surface wave and structural vibration analysis.
- **Key Features**:
    - **Natural frequency:** 10 Hz ± 5%
    - **Sensitivity:** 28.8 V/m/s typical
    - **Coil resistance:** ~375 Ω
    - **Orientation:** horizontal (X/Y pair)
    - **Operating range:** −40 °C to +100 °C

    - **Housing:** robust molded plastic, field-sealable, ~25 mm × 36 mm per axis module
- **Good for INS or vibration experiments because**: allows **horizontal ground motion measurement**, complements vertical SM-24; commonly used in 3-component seismic strings.
- **Key Links**:
    - Datasheet (PDF): Sercel SM-4 Geophone Specification Sheet
    - Product overview: Sercel Land Acquisition Sensors

EG-4.5-II Geophone (GeoSpace Technologies)

- **Type / Sensor Suite**: **High-sensitivity 4.5 Hz** low-frequency analog geophone.
- **Key Features**:
    - **Natural frequency:** 4.5 Hz ± 5%
    - **Coil resistance:** ~375 Ω ± 5%
    - **Sensitivity:** 28.8 V/m/s typical
    - **Damping:** 0.7 ± 5%
    - **Operating temperature:** −40 °C to +100 °C
    - **Distortion:** < 0.2% at 0.7 in/s p-p
    - **Dimensions:** ~25 mm diameter × 36 mm height
- **Good for low-frequency detection because**: extended bandwidth captures **lower vibration frequencies** than 10 Hz elements like the SM-24; widely used in **micro-tremor**, **impact**, and **seismic refraction** testing.
- **Key Links**:
    - Datasheet (PDF): GeoSpace EG-4.5-II Element Specification Sheet
    - Product page: GeoSpace Technologies Sensors

[11a] Our key component for our handheld device will be a radar module. Can you talk to me about some options to consider?

[11b] Infineon BGT60TR13C

- **Type / Sensor Suite**: 60 GHz (mmWave) radar transceiver (single-chip) in wafer-level packaging for motion/occupancy detection.
- **Key Features**:
    - Operating frequency: ~57–64 GHz (ISM band) with built-in PLL, mixers, and antenna interfaces. (infineon.com)
    - Small form factor: wafer-level chip scale; typically mounted on evaluation PCB.
    - Supports presence detection, people counting, block detection, small movement / vital signs (micro-Doppler) functionalities. (infineon.com)
    - Low power: typical power consumption ~40–60 mW depending on mode.

- **Good for your project because**: compact radar sensor ideal for integration into robotics or INS systems for obstacle detection / motion sensing in addition to GNSS/INS.
- **Key Links**:
  - Datasheet (PDF): https://www.infineon.com/dgdl/Infineon-BGT60TR13C-DataSheet-v01_00-EN.pdf (infineon.com)
  - Product page: Infineon BGT60TR13C. (infineon.com)

Infineon BGT60ATR24C

- **Type / Sensor Suite**: 60 GHz (mmWave) antenna-integrated radar transceiver module (evaluation board) for motion and gesture sensing.
- **Key Features**:
  - Operating frequency: ~57-64 GHz, multi-antenna; supports motion detection & direction of arrival (DoA) features. (infineon.com)
  - Board evaluation module with built-in antennas, SMA connectors for external antennas.
  - Designed for smart home / IoT radar use-cases: people detection, gesture control, occupancy.
  - Typical power supply: 3.3 V, consumption ~60–70 mW in typical mode.
- **Good for your project because**: off-the-shelf radar module gives you quick integration for motion sensing or relative speed/direction measurement to complement GNSS/INS.
- **Key Links**:
  - Datasheet (PDF): https://www.infineon.com/dgdl/Infineon-BGT60ATR24C-DataSheet-v01_00-EN.pdf (infineon.com)
  - Product page: Infineon BGT60ATR24C. (infineon.com)

Texas Instruments IWR6843AOP EVM

- **Type / Sensor Suite**: 60 GHz mmWave radar evaluation module with integrated antennas and DSP, for advanced sensing (gesture, people, vital-sign) and sub-cm motion detection.
- **Key Features**:
  - Tri-chip mmWave sensor: includes DSP + RF + antennas in evaluation board form.
  - Multi-object tracking, Doppler measurement, range resolution ~4 cm, frame rate configurable. (ti.com)
  - SPI/PCIe host interface, onboard microcontroller for preprocessing, SDK support in TI mmWave SDK.
  - Supply: 5 V via USB for evaluation board; module itself ~3.3 V; power consumption variable depending on mode (~1 W in full mode).

- **Good for your project because**: high-level radar module with rich sensing capabilities—allows you to prototype not just motion but full environment sensing, which can enhance the GNSS/INS system.
- **Key Links**:
  - Datasheet (PDF): https://www.ti.com/lit/ds/symlink/iwr6843opm.pdf (ti.com)
  - Product page: IWR6843AOP EVM. (ti.com)

Texas Instruments AWR1843BOOST EVM

- **Type / Sensor Suite**: 77 GHz mmWave radar evaluation module (automotive grade) integrated antennas, DSP, for high-resolution motion and object detection.
- **Key Features**:
  - Operating frequency: 77-81 GHz (automotive radar bands); excellent range/detection resolution for automotive, robotics, navigation support. (ti.com)
  - On-board radar chip + antenna array + MCU + USB interface; capable of range resolution ~4 cm, speed measurement, object tracking.
  - Supply: typically 5 V via USB for eval board; module internal supply ~3.3 V; active power ~1 W+.
- **Good for your project because**: highest frequency radar here—best for high-resolution detection of smaller objects, speed measurement, close-range sensing that can feed into inertial navigation or obstacle avoidance.
- **Key Links:**
  - User guide (PDF): https://www.ti.com/lit/ug/spruivd/spruivd.pdf (ti.com)
  - Product page: AWR1843BOOST EVM. (ti.com)

[12a] Talk to me about infrared thermal sensors. What are some models to compare?

[12b] Panasonic AMG8833

- **Type / Sensor Suite**: 8×8 pixel thermal infrared sensor array (digital output)
- **Key Features**:
  - 64 pixel (8×8) thermopile array. Panasonic Industrial+2EcoSensors+2
  - I²C interface; includes interrupt output for pixel threshold crossing. Adafruit Learn+1
  - Typical supply: 3.3 V. DigiKey+1
  - Field of view: ~60° typical. Adafruit Learn+1
  - Temperature measurement range for thing being measured (object) ~0–80 °C for the standard version. cdn.sparkfun.com+1

- **Good for INS/thermal-fusion because**: Low-cost thermal imaging array gives you a rough “heat-map” view—good for detecting people, hot zones, movement when combined with GNSS/INS data.
- **Key Links**:
  - Datasheet: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-amg8833-8x8-thermal-camera-sensor.pdf Adafruit Learn+1
  - Product page: https://industrial.panasonic.com/ww/products/pt/grid-eye/models/AMG8833 Panasonic Industrial
  - Distributor page with specs: https://www.digikey.com/en/products/detail/panasonic-electronic-components/AMG8833/5825302 DigiKey

Melexis MLX90640

- **Type / Sensor Suite**: 32×24 pixel infrared thermal sensor array (far-IR)
- **Key Features**:
  - 768 pixels (32×24) array. Melexis+1
  - I²C digital interface. Melexis+1
  - Supply: ~3.3 V; current consumption less than 23 mA typical. Melexis+1
  - Field of view options: ~55°×35° or 110°×75°. Melexis+1
  - Noise equivalent temperature difference (NETD): ~0.1 K RMS @1 Hz refresh. Melexis
- **Good for INS/thermal-fusion because**: Higher resolution thermal array compared to 8×8. Good for more detailed heat pattern detection, potentially aligning with location/trajectory from INS.
- **Key Links**:
  - Datasheet: https://www.melexis.com/-/media/files/documents/datasheets/mlx90640-datasheet-melexis.pdf Melexis+1
  - Example breakout & specs: https://makersportal.com/shop/mlx90640-thermal-camera-for-raspberry-pi-32-x-24-pixels Maker Portal

FLIR Lepton 3.5

- **Type / Sensor Suite**: Radiometric long-wave infrared (LWIR) micro-thermal camera module (160×120 pixels)
- **Key Features**:
  - 160×120 pixel array, uncooled microbolometer. Mouser Electronics+1
  - Output: calibrated temperature values (radiometric). cdn.sparkfun.com+1
  - Scene temperature range up to +400 °C in low gain mode (for Lepton 3.5). DigiKey+1

    - Supply and digital/video interface (SPI/VoSPI) for camera module. GroupGets Files
- **Good for INS/thermal-fusion because**: Very high resolution compared to simple arrays; gives detailed thermal imagery and temperature info which can be fused with navigation/positioning data for advanced situational awareness (e.g., thermal mapping, people detection, environment scanning).
- **Key Links**:
    - Engineering datasheet: https://cdn.sparkfun.com/assets/f/6/3/4/c/Lepton_Engineering_Datasheet_Rev200.pdf cdn.sparkfun.com+1
    - Product info page: https://oem.flir.com/en-hk/products/lepton?segment=oem&vertical=microcam oem.flir.com

Melexis MLX90614

- **Type / Sensor Suite**: Single (or dual) zone non-contact infrared thermometer sensor
- **Key Features**:
    - Factory-calibrated IR thermometer; object temperature range from roughly -70 °C to +382.2 °C (depending version). Illinois Physics Courses+1
    - Interface: SMBus (I²C) and/or PWM output. Melexis
    - Resolution: ~0.02 °C via I²C, standard accuracy ~0.5 °C at 0-50 °C ambient. Components101+1
    - Package: TO-39 can (for single-zone) or module, small size. Melexis
- **Good for INS/thermal-fusion because**: Simple, cost-effective thermal measurement—good for detecting a hot object or measuring surface temperature where full thermal image isn't required, can complement position data for hazard detection or temperature monitoring.
- **Key Links**:
    - Datasheet: https://www.melexis.com/-/media/files/documents/datasheets/mlx90614-datasheet-melexis.pdf Melexis
    - Product overview: https://components101.com/sensors/melexis-mlx90614-contact-less-ir-temperature-sensor Components101

[13a] Due to the radar sensor needing to be compliant with FCC regulations pertaining to distancing, we are thinking about incorporating some sort of motion detection to suspend transmission from the radar module if someone gets within a certain distance. Can you provide me with some options to accomplish this? One idea was an USS.

[13b] HC-SR04 Ultrasonic Distance Sensor

- **Type / Sensor Suite:** Ultrasonic time-of-flight (ToF) distance sensor.
- **Key Features:**
  - Measurement range: **2 cm – 400 cm** (effective up to ~3 m for reliable results).
  - Accuracy: ±3 mm.
  - Operating frequency: **40 kHz**.
  - Interface: **Trigger** and **Echo** pins for pulse-width timing.
  - Supply: **5 V** DC, average current ~15 mA.
  - Output: TTL pulse proportional to distance (t_echo × speed_of_sound / 2).
  - Operating temperature: 0 °C – 70 °C.
- **Good for your project because:** Simple, low-cost ranging for obstacle detection and proximity-based radar activation. Excellent for short-range fusion with IR or radar to confirm object presence.
- **Key Links:**
  - Datasheet (PDF): HC-SR04 Ultrasonic Sensor Datasheet
  - Product overview: SparkFun HC-SR04 Product Page
  - Arduino tutorial/reference: Arduino HC-SR04 Guide

HC-SR501 Passive Infrared (PIR) Motion Sensor

- **Type / Sensor Suite:** Pyroelectric infrared (motion) detector.
- **Key Features:**
  - Detects human motion using changes in IR radiation levels.
  - Detection range: **up to 7 m** (adjustable using onboard potentiometers).
  - Adjustable **delay time (5 s – 5 min)** and **sensitivity** controls.
  - Output: **Digital HIGH/LOW** logic (trigger output).
  - Field of view: **~110° × 70°**.
  - Supply: **4.5 V – 20 V**, typical current ~50 µA.
- **Good for your project because:** Acts as a low-power motion trigger for radar or thermal sensors; ideal for human detection activation or standby wake systems.
- **Key Links:**
  - Datasheet (PDF): HC-SR501 PIR Sensor Specification
  - Product overview: Adafruit HC-SR501 Product Page
  - Example integration: Arduino Motion Sensor Guide

ST VL53L0X Time-of-Flight Laser Ranging Sensor

- **Type / Sensor Suite: Laser-based proximity (ToF) sensor** using VCSEL + SPAD receiver.
- **Key Features:**
  - Operating principle: **FlightSense™ ToF** — measures light travel time directly.
  - Range: **up to 2 m** (depends on target reflectivity and lighting).

    - Accuracy: ±3 % typical in 50 cm range.
    - Interface: **I²C** (up to 400 kHz).
    - Supply voltage: **2.6 V – 3.5 V**, typical current ~20 mA.
    - Field of view: **25°**.
    - Measurement rate: **up to 50 Hz**.
- **Good for your project because:** Compact, precise short-range sensor with millimeter accuracy — ideal for fine distance fusion with radar/IR for confirming targets or calibrating distance-to-human detection.
- **Key Links:**
    - Datasheet (PDF): VL53L0X Datasheet
    - Product page: STMicroelectronics VL53L0X Product Overview
    - Example breakout: Adafruit VL53L0X Time of Flight Sensor

[14a] I am doing some research on a user interface for a handheld device. We want to use LEDs, buttons, and a buzzer or speaker. Talk to me about my options.

[14b] Pushbuttons and User Interface Components
APEM IPR1SAD2 Pushbutton

- **Type / Function:** Momentary tactile switch with sealing and integrated LED ring illumination.
- **Key Features:**
    - IP67 rated for dust and water protection.
    - Stainless-steel actuator; ring-illumination for status feedback.
    - Contact rating — 2 A @ 48 V DC (max).
    - Lifetime: 1 million cycles.
    - Operating force ≈ 3 N; travel ≈ 1.5 mm.
    - Mounting: panel cut-out 19 mm.
- **Good for project:** Durable sealed design ensures reliability in outdoor or rescue conditions; LED backlight can double as power/confirm indicator.
- **Key Links:**
    - APEM IPR1SAD2 Datasheet
    - Mouser Product Page

E-Switch PV6F240SS-331C Pushbutton

- **Type / Function:** Illuminated anti-vandal metal pushbutton.
- **Key Features:**
    - Stainless steel, IP67 sealed.
    - Momentary (normally open) configuration.
    - Built-in LED ring indicator (3.3 V DC, 20 mA).
    - Electrical life: 50 000 cycles.
    - Mechanical life: 1 000 000 cycles.
    - Contact rating — 2 A @ 36 V DC.

- **Good for project:** Provides power and mode-select control with integrated visual feedback. Excellent tactile feedback for gloved users in the field.
- **Key Links:**
  - E-Switch PV6 Series Datasheet
  - Digi-Key Product Page

C&K PTS645VL58-2 LFS Tactile Switch

- **Type / Function:** Compact tactile switch for input control (reset/mark).
- **Key Features:**
  - Momentary SPST NO type.
  - Low profile (4.3 mm height); actuation force 160 gf.
  - Rating — 50 mA @ 12 V DC.
  - Mechanical life — 200 000 cycles.
  - Compact 6×6 mm footprint suitable for PCB mounting.
- **Good for project:** Ideal for small, low-voltage functions like “Mark Victim” or “System Reset.”
- **Key Links:**
  - C&K PTS645 Datasheet
  - Mouser Product Page

LED Indicators

- **Selected Type:** High-brightness 3 mm or 5 mm diffused red/green LEDs.
- **Specs:**
  - Forward voltage: 2.0 V (Red) / 3.0 V (Green).
  - Forward current: 10–20 mA typical.
  - Typical luminance: 1000–3000 mcd.
  - Lifetime: > 100 000 h.
- **Good for project:** Simple visual feedback—red for “Human Detected”, green for “Ready/Active”.
- **Reference Links:**
  - Kingbright WP710A10SRD LED Datasheet
  - Vishay TLHR5400 Product Page

Piezo Buzzer – CUI CPT-1127C-C3035

- **Type / Function:** Magnetic piezo buzzer (active drive).
- **Key Features:**
  - Rated voltage 3 V DC (active).
  - Sound pressure ≈ 85 dB @ 10 cm.
  - Resonant frequency ≈ 3.4 kHz.
  - Current consumption ≈ 20 mA.
  - Diameter 12 mm; height 7 mm.

- **Good for project:** Compact, low-power audible alert for detection events; can be driven directly from MCU GPIO with transistor stage.
- **Key Links:**
  - CUI Buzzer Datasheet
  - Digi-Key Product Page

Speaker – CUI CDS-15135-SP Mini Speaker

- **Type / Function:** Small dynamic speaker for audio feedback.
- **Key Features:**
  - Rated power 0.5 W, max 0.8 W.
  - Impedance 8 Ω.
  - Frequency range 500 Hz – 6 kHz.
  - Sound pressure ≈ 87 dB @ 10 cm.
  - Dimensions 15 × 3.5 mm.
- **Good for project:** Used for richer audio alerts or voice output; complements the piezo buzzer for multi-tone notifications.
- **Key Links:**
  - CUI CDS-15135-SP Datasheet
  - Mouser Product Page

Display Screen – SSD1963 4.3-inch TFT

- **Type / Function:** 480 × 272 pixel TFT LCD with capacitive touch.
- **Key Features:**
  - 16.7 M colors (TFT RGB 888).
  - Parallel 8080 interface / SPI controller mode.
  - Supply voltage 3.3 V logic, 5 V backlight.
  - Controller IC: SSD1963 with internal frame buffer.
- **Good for project:** Central user interface for displaying detection heatmaps, coordinates, and system diagnostics.
- **Key Links:**
  - EastRising TFTM043A1-7S Datasheet
  - SSD1963 Controller Spec

✅ Summary Table – UI Components

Component	Type	Rated Voltage	Current	Special Feature	Use
APEM IPR1SAD2	Sealed pushbutton	3–5 V LED, ≤ 48 V contact	20 mA LED	IP67 metal ring LED	Power / Mode
E-Switch PV6F240SS-331C	Anti-vandal PB	3.3 V LED	20 mA	Stainless IP67	Power / Reset

Component	Type	Rated Voltage	Current	Special Feature	Use
C&K PTS645VL58-2 LFS	Tactile PB	3.3 V	< 50 mA	Compact PCB mount	Mark/Reset
LED Indicator	Diffused LED	2–3 V	10–20 mA	High brightness	Visual alert
Piezo Buzzer (CUI)	Active buzzer	3 V	20 mA	85 dB output	Audible alert
Speaker (CUI)	Mini speaker	3.3 V	~100 mA	8 Ω impedance	Audio feedback
LCD (SSD1963)	4.3" TFT touch	3.3 V logic / 5 V BL	~150 mA	Capacitive touch	Main display UI

[15a] I need some LCD options for a project I am working on. This LCD needs to be practical in emergency situations, for first responders.

[15b] Display Modules (LCD / TFT)
ILI9341 TFT (2.4"–3.2" typical)

- **Type / Display Driver:** TFT LCD controller with 240 × 320 pixel resolution (QVGA).
- **Key Features:**
  - 262 K color (RGB 565) support.
  - Parallel (8080/6800) and SPI interfaces.
  - Supply voltage: 2.8 V–3.3 V logic; 5 V tolerant inputs (some breakouts).
  - Frame memory: 240 × 320 × 18-bit internal buffer.
  - Typical power: ~60–90 mA (display on, 3.3 V).
  - Compatible with **Adafruit ILI9341**, **ESP32**, and **Arduino** drivers.
- **Good for your project:** Ideal compact screen for debugging or secondary UI. Stable, low-cost, and widely supported.
- **Key Links (for references):**
  - ILI9341 Datasheet (Solomon Systech)
  - Adafruit ILI9341 Product Page

RA8875 TFT Controller Board

- **Type / Display Driver:** TFT LCD controller with integrated graphics acceleration.
- **Key Features:**
  - Supports TFT panels up to **800 × 480** (WVGA).
  - 2D drawing engine: line, rectangle, circle, text hardware acceleration.

    - Integrated **touch controller** (resistive).
    - Interfaces: SPI, 8-bit or 16-bit parallel, PWM backlight control.
    - Supply: 3.3 V logic; 5 V tolerant inputs.
- **Good for your project:** Great when using larger displays—reduces MCU workload with built-in hardware graphics and touch handling.
- **Key Links:**
    - RA8875 Datasheet (RAiO Technology)
    - Adafruit RA8875 Controller Board

RA8875 with Capacitive Touch (4.3"–7")

- **Type / Display Module:** RA8875-driven TFT with built-in **capacitive touch controller**.
- **Key Features:**
    - Resolution: typically **480 × 272** (4.3") or **800 × 480** (5"–7").
    - Uses **RA8875** for display control + **FT5x06** series touch controller.
    - Interfaces: SPI/Parallel for display, I²C for touch.
    - Brightness: 500–700 nit typical.
    - Capacitive multi-touch supported (1–5 points).
- **Good for your project:** Adds modern capacitive interface for field operation; reduces mechanical wear compared to resistive touch.
- **Key Links:**
    - RA8875 + Capacitive Touch Product Example (BuyDisplay)
    - FT5x06 Touch Controller Datasheet

4.3" 800 × 480 IPS TFT

- **Type / Display Module:** 4.3-inch **In-Plane Switching (IPS)** TFT with full-color display.
- **Key Features:**
    - Resolution: **800 × 480 (WVGA)**.
    - Viewing angle: **80°/80°/80°/80°** (true IPS).
    - Brightness: **400–500 nit** typical.
    - Interface: 24-bit RGB (parallel), supports external controllers (RA8875/SSD1963).
    - Supply: 3.3 V logic; 5 V for backlight.
    - Operating temperature: −20 °C to +70 °C.
- **Good for your project:** High-contrast IPS display ensures clear visibility at all angles—ideal for outdoor handheld use.
- **Key Links:**
    - ER-TFT043-4 Datasheet (BuyDisplay)
    - Newhaven 4.3" IPS LCD Product Page

4.3" 740-Nit Sunlight Readable TFT

- **Type / Display Module:** High-brightness TFT LCD for outdoor applications.
- **Key Features:**
  - Resolution: 480 × 272 (typ.) or 800 × 480 (depending model).
  - Brightness: **740 nit** (sunlight-readable).
  - Contrast ratio: ~800:1.
  - Interface: 24-bit RGB, parallel; supports SSD1963/RA8875.
  - Operating temp: −30 °C to +85 °C.
- **Good for your project:** Designed for outdoor readability—perfect for rescue, field, or search-and-rescue handheld displays.
- **Key Links:**
  - ER-TFT043-7 Datasheet (BuyDisplay)
  - Lumineq Outdoor TFT Overview

4.3" IPS Capacitive Touch TFT

- **Type / Display Module:** Integrated **IPS LCD + capacitive touch panel** (CTP).
- **Key Features:**
  - Resolution: 480 × 272 or 800 × 480 (depending model).
  - Viewing angle: 80° all directions.
  - Brightness: 500 nit typical.
  - Touch interface: I²C (CTP).
  - Display interface: 24-bit RGB, SPI, or 8080-parallel.
  - Supply: 3.3 V logic, 5 V backlight.
- **Good for your project:** Combines clear IPS visibility with a modern capacitive interface.
- **Key Links:**
  - ER-TFT043-5CP Datasheet (BuyDisplay)
  - FT5336 Touch Controller Datasheet (Capacitive Touch IC)

✅ Summary Table – Display Modules

Display	Size	Resolution	Brightness	Interface	Touch	Notes
ILI9341	2.4"–3.2"	240×320	~250 nit	SPI / 8080	None	Compact debug screen
RA8875 TFT	Up to 7"	800×480	400–500 nit	SPI / Parallel	Resistive	Graphics acceleration
RA8875 + CTP	4.3"–7"	480×272 – 800×480	500–700 nit	SPI / I²C	Capacitive	Multi-touch, modern UI
4.3" IPS TFT	4.3"	800×480	400–500 nit	RGB / Parallel	Optional	Wide viewing angle
4.3" 740-nit TFT	4.3"	800×480	740 nit	RGB / Parallel	Optional	Sunlight readable
4.3" IPS CTP	4.3"	800×480	500 nit	RGB + I²C	Capacitive	Best all-around handheld UI

[16a] I have several sensors needing to be interfaced. The TI IWR6842AOP EVM, DWM300 EVM, M135 INS, an IR, buttons, LEDs, a ILI9341 LED screen. I need to explore MCU options.

[16b] Raspberry Pi 4 Model B

- **Processor:** Broadcom BCM2711, Quad-Core Cortex-A72 (ARM v8) @ 1.5 GHz
- **Memory:** 2 GB / 4 GB / 8 GB LPDDR4
- **Storage:** microSD slot (up to 128 GB)
- **Connectivity:** Wi-Fi 802.11ac, Bluetooth 5.0, Gigabit Ethernet
- **Interfaces:** 40 GPIO pins, 4 × USB ports (2 × USB 3.0, 2 × USB 2.0), HDMI × 2, CSI, DSI
- **Operating Voltage:** 5 V DC via USB-C (3 A typical)
- **Power Consumption:** 2.7–6.4 W depending on load
- **Operating System:** Raspberry Pi OS, Ubuntu, Linux-based
- **Dimensions:** 85 × 56 × 17 mm
- **Cost:** $35–$75 depending on RAM size
- **References:**
  - Raspberry Pi 4 Product Page
  - Broadcom BCM2711 Datasheet

STM32H7 Series (Example: STM32H743VI)

- **Processor:** ARM Cortex-M7 @ 480 MHz
- **Flash Memory:** 2 MB
- **RAM:** 1 MB SRAM
- **Peripherals:** 3 × ADCs (16-bit), 2 × DAC, CAN, SPI, I²C, UART, USB OTG, Ethernet MAC, SDMMC
- **Operating Voltage:** 1.8–3.6 V
- **Power Consumption:** 15–40 mA typical at 3.3 V
- **Temperature Range:** −40 °C to +85 °C
- **Package Options:** LQFP100, BGA
- **Dimensions:** varies by package (10×10 mm typical)
- **Cost:** $8–$15 (depending on series)
- **References:**
  - STM32H7 Product Page
  - STM32H743VI Datasheet

ESP32-WROOM-32E

- **Processor:** Dual-Core Xtensa LX6 @ 240 MHz
- **Flash Memory:** 4 MB (options up to 8 MB)
- **RAM:** 520 KB SRAM
- **Wireless:** Wi-Fi 802.11 b/g/n, Bluetooth v4.2 BR/EDR & BLE
- **Interfaces:** 3 × UART, 3 × SPI, 2 × I²C, 2 × DAC, 12-bit ADC, PWM, CAN

- **Operating Voltage:** 3.0–3.6 V
- **Power Consumption:**
  - Active: ~160 mA
  - Light Sleep: ~0.8 mA
  - Deep Sleep: ~10 µA
- **Operating Temp:** −40 °C to +85 °C
- **Dimensions:** 18 × 25.5 × 3 mm
- **Cost:** $3–$6 per module
- **References:**
  - ESP32-WROOM-32E Datasheet
  - Espressif ESP32 Overview

ESP32-S3-WROOM-1

- **Processor:** Dual-Core Xtensa LX7 @ 240 MHz
- **Flash Memory:** 4–8 MB
- **RAM:** 512 KB SRAM + optional 8 MB PSRAM
- **Wireless:** Wi-Fi 802.11 b/g/n, Bluetooth 5 (LE + Mesh)
- **Interfaces:** USB OTG, SPI, I²C, I²S, UART, CAN, PWM, ADC, DAC
- **AI Acceleration:** Vector instructions for ML, speech, or FFT
- **Operating Voltage:** 3.0–3.6 V
- **Power Consumption:** ~180 mA active, 10 µA deep sleep
- **Operating Temp:** −40 °C to +85 °C
- **Dimensions:** 18 × 25.5 × 3 mm
- **Cost:** $5–$8
- **References:**
  - ESP32-S3-WROOM-1 Datasheet

ESP32-S6-WROOM-1 (Upcoming)

- **Processor:** Dual-Core RISC-V @ 240 MHz
- **Flash Memory:** 4–16 MB
- **RAM:** 512 KB SRAM + external PSRAM
- **Wireless:** Wi-Fi 6 (802.11ax) + Bluetooth 5 LE
- **Interfaces:** UART, SPI, I²C, PWM, USB 2.0, Ethernet MAC
- **Operating Voltage:** 3.0–3.6 V
- **Power Consumption:** ~130 mA active (est.), <10 µA deep sleep
- **Security:** Secure boot, hardware RSA/AES acceleration
- **Dimensions:** 18 × 25.5 mm (same form factor as ESP32-S3)
- **Cost:** Estimated $6–$9
- **References:**
  - ESP32-S6 Launch Info

Raspberry Pi Pico (optional baseline reference)

- **Processor:** RP2040 Dual-Core ARM Cortex-M0+ @ 133 MHz

- **Flash Memory:** 2 MB QSPI
- **RAM:** 264 KB SRAM
- **Interfaces:** 2 × UART, 2 × SPI, 2 × I²C, PWM, ADC, PIO (Programmable I/O)
- **Operating Voltage:** 1.8–5.5 V
- **Power Consumption:** ~91 mA active
- **Dimensions:** 21 × 51 mm
- **Cost:** $4
- **References:**
    - Raspberry Pi Pico Datasheet

[17a] Can you now make a realistic rendering of a person holding it and the tablet is facing towards a crumbled building? You can leave out their face and maybe just have their hands and the back of their head. Maybe in fire fighter or first responder gear.

[17b] Got it 👍 since you want your **Portable AI Enhanced Life Detector** to look somewhat like an iPad, I can draft you a **blueprint-style rendering** (a prototype concept) that shows:

- A tablet-style device with a **touchscreen interface**.
- Indicators for **sensor inputs** (camera, mic, thermal, etc.).
- A **side-mounted button** (power/reset).
- A **status LED bar** (e.g., green for normal, red for alert).
- Ports (USB-C for charging, maybe expansion).
- Software interface mockup: map view, detection alerts, data overlay.

Would you like me to generate:

1. **A front + side blueprint diagram** (technical style, labels for components),
2. **A 3D-style concept render** (like a sleek prototype marketing visual),

or both?

[17c] Option number 1.

[17d] Figure 4 was rendered.

[18a] Can you now make a realistic rendering of a person holding it and the tablet is facing towards a crumbled building? you can leave out their face and maybe just have their hands and the back of their head. Maybe in fire fighter or first responder gear.

[18b] Figure 5 was rendered.