

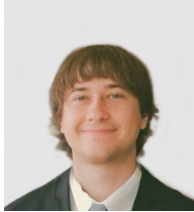
RADIO DIVERSITY AUTOMATION

Divide and Conquer Document

Department of Electrical Engineering and Computer Science

University of Central Florida Dr. Chung Yong Chan Dr. Lei Wei Dr. Aravinda Kar Group 26

Authors:



Evan Crawford

*Electrical
Engineering*



Jurgis Siusys

*Computer
Engineering*



Kim Hoang

*Computer
Engineering*

Reviewer Committee:

Dr. Saleem Shawneh

Department of ECE

Associate Professor

Dr. Qun Zhou Sun

Department of ECE

Associate Professor

Dr. Mike Borowczak

Department of ECE

Associate Professor

Dr. Mark Maddox

Department of ECE

Associate Professor



Table of Contents

Chapter 1 Executive Summary.....	1
Chapter 2 Project Description.....	2
2.1 Motivation and Background.....	2
2.2 Existing Products.....	2
2.2.1 General Applications.....	3
2.2.2 Proprietary Applications.....	3
2.2.3 Comparison of Our Design to Available Products.....	3
2.3.1 Basic Goals.....	4
2.3.2 Advanced Goals.....	4
2.3.3 Stretch Goals.....	5
2.3.4 Objectives.....	5
2.4 Features and Functionalities.....	6
2.4.1 Design Implementations.....	6
2.4.2 Customer Input.....	6
2.5 Specifications.....	7
2.5.1 System Specs.....	7
2.5.2 Component Specs.....	7
2.6 House of Quality.....	8
2.7 Block Diagrams.....	10
2.7.1 Hardware Block Diagram of Node Array.....	10
2.7.2 Software Flow Chart.....	11
Chapter 3 Research and Investigation.....	12
3.1 Hardware Technologies.....	12
3.1.1 Central Processor.....	12
3.1.1.1 MPU vs. SBC.....	12
3.1.1.1.1. Pros and Cons of SBCs.....	12
3.1.1.1.2. Pros and Cons of MCUs.....	13
3.1.1.1.3. Why SBC could be a best fit?.....	13
3.1.1.1.4. Choosing between MCU and SBC.....	14
3.1.1.2 SBC Selection.....	16
3.1.1.2.1. Raspberry Pi Series.....	16
3.1.1.2.2 Raspberry Pi Zero (W).....	18
3.1.1.2.3 Raspberry Pi Compute Module Series.....	18
3.1.1.2.4. Qualcomm Snapdragon.....	19
3.1.1.2.5. NXP i.MX.....	20
3.1.1.2.6. Intel Atom, Core, and Xeon.....	21
3.1.1.2.7. Texas Instruments Sitara.....	22
3.1.1.2.8. SBC Comparison.....	23

3.1.1.3 SBC Selection.....	24
3.1.2. Antenna.....	25
3.1.2.1 Choosing the Antenna.....	25
3.1.2.1.1. Log-Periodic Dipole Array (LPDA) Antennas.....	25
3.1.2.1.2. Yagi-Uda Antenna.....	26
3.1.2.1.3. Dipole Antennas.....	26
3.1.2.1.4. Phased Array Antennas.....	26
3.1.2.1.5. Monopole Antennas.....	26
3.1.2.1.6. Microstrip Patch Antennas.....	27
3.1.2.1.7. Antenna Comparison.....	27
3.1.2.2 Software Defined Radios.....	28
3.1.2.3 Setup with SDRs and Antennas.....	29
3.1.2.4 Antenna Selection.....	30
3.1.3 Software Defined Radio.....	30
3.1.3.1 Choosing an SDR.....	30
3.1.3.1.1 Multiple Single Input SDRs.....	31
3.1.3.1.2 Multi Input SDR.....	31
3.1.3.2 Connection to Raspberry Pi.....	32
3.1.3.3 Final Selection.....	32
3.1.4 Raspberry Pi Touchscreen.....	33
3.1.4.1 Choosing a Touchscreen.....	33
3.1.4.2 Integration with Raspberry Pi.....	34
3.1.4.3 GUI Design and Layout.....	34
3.1.4.4 Hardware Options.....	35
3.1.4.5 Final Selection.....	36
3.2. Power Supply System.....	37
3.2.1 Batteries.....	38
3.2.1.1 Choosing a Battery.....	38
3.2.1.2 Available Battery Options.....	38
3.2.1.3 Battery Option Chart.....	39
3.2.1.4 LiFePO4 Battery Choices.....	40
3.2.2 Solar Panels.....	41
3.2.2.1 General Types of Panels.....	41
3.2.2.2 Pros and Cons.....	41
3.2.2.3 Specific Models.....	43
3.2.3 Charge Controllers.....	44
3.2.3.1 Purpose of Charge Controller.....	44
3.2.3.2 Types of Charge Controllers.....	44
3.2.3.3 Charge Controller Options.....	44
3.2.4 Wiring Setup.....	45
3.2.4.1 For a Single Unit.....	45

3.2.4.2 For Multiple Unit.....	45
3.2.4.3 Additional Recommendations.....	45
3.2.5 Power Supply Part Selection.....	45
3.3 Communication Technologies.....	46
3.3.1. Broadcast Methods.....	46
3.3.1.1 Zigbee.....	46
3.3.1.2 Z-Wave.....	48
3.3.1.3 Near Field Communication (NFC).....	49
3.3.1.4 Bluetooth.....	50
3.3.1.5 Wi-Fi.....	51
3.3.1.6 LoRa.....	52
3.3.3 Final Broadcast Selection.....	54
3.4 Software Technologies.....	55
3.4.1. MATLAB.....	55
3.4.2 Node.js.....	56
3.4.3 Java.....	56
3.4.4 Python.....	57
3.4.5 Language selection.....	57
3.5 Development Environment.....	58
3.5.1 Thonny.....	58
3.5.2 PyCharm.....	59
3.5.3 Jupyter Notebook.....	59
3.5.4 Raspberry Pi OS (with IDLE).....	59
3.5.5 Anaconda (Spyder).....	60
3.6 Data Transfer Mechanism.....	61
3.7 Data Transmission Process.....	62
3.8 Data Structure and Storage.....	63
Chapter 4 Standards and Design Constraints.....	64
4.1 Software.....	64
4.1.1 Software Standards.....	64
4.1.1.1 IEEE 12207 – Software Life Cycle Processes.....	64
4.1.1.2 ISO/IEC 27001 – Information Security Management.....	66
4.2 Hardware.....	67
4.2.1 Hardware Standards.....	68
4.2.2 Hardware Design Constraints.....	69
4.2.3 Non-Technical Constraints.....	70
Environmental Constraints.....	72
Chapter 5. Comparison of ChatGPT with Similar Platforms.....	74
5.1 Case study 1: ChatGPT vs Microsoft Copilot.....	75
5.2 Case study 2: ChatGPT vs Google Gemini.....	75

5.3 Case study 3: ChatGPT vs Grok.....	76
Chapter 6. Hardware Design.....	77
6.1 Power Supply System.....	77
6.2 Broadcast & Receiver RF Subsystem.....	78
6.2.1 Broadcast Hardware.....	78
6.2.2 Antenna Schematic (Active/Passive).....	79
6.2.2 Spectrum Analyzer Connections.....	80
6.3 Data Processing and Output System.....	80
6.3.1 Raspberry Pi.....	80
6.3.2 Touchscreen User Interface.....	81
Chapter 7. Software Design.....	81
7.1 Use Case Diagram.....	81
7.5. User Interface (UI) Design.....	85
Dashboard Functionality.....	88
Signal Panels.....	88
Spectrum Plot.....	89
Status Log Section.....	89
Control Panel.....	89
Chapter 8: Prototyping and Testing.....	90
8.1 Hardware Testing.....	90
8.2 Software Testing.....	91
8.3 Integration Testing.....	92
8.4 Enclosure and Environmental Housing.....	93
Chapter 9: Test Plan.....	94
9.1 Raspberry Pi as the Target Hardware Platform.....	95
9.2 Main Unit Testing (Central Raspberry Pi).....	95
9.3 Diversity Logic and Selection Testing.....	96
9.4 UI and Responsiveness Testing.....	96
9.5 Full System Testing.....	96
9.6 Summary.....	97
Chapter 10. Administrative Content.....	98
10.1 Project Budget and Financing.....	98
10.2 Bill of Materials.....	98
10.3 Work Distribution.....	99
10.4 Schedule and Milestones.....	100
10.4.1 Senior Design 1.....	100
10.4.2 Senior Design 2.....	100
Chapter 11: Conclusion.....	102
APPENDIX A REFERENCES.....	105
APPENDIX B.....	110
5.1. ChatGPT vs Copilot.....	110

5.2 Case Study - ChatGPT.....	113
Step 1: Solar Panel to MPPT Charge Controller.....	114
Step 2: Battery Connection.....	114
Step 3: Voltage Regulation for Each Component.....	114
12V to 5V Conversion (For Raspberry Pi, Samsung S24, LNA, etc.).....	114
Direct 12V Connection (For Signal Hound BB60D).....	114
5.3 ChatGPT vs Grok.....	116
7.4 ERD Overview.....	122
APPENDIX C - Code.....	124

List of Figures

Figure 2.1 House of Quality Diagram	9
Figure 2.2 Hardware Block Diagram of System	10
Figure 2.3 Software block diagram	11
Figure 6.1 Power Subsystem Wiring Diagram	78
Figure 6.2 Passive Antenna Setup	79
Figure 6.2.2 Spectrum Analyzer	80
Figure 7.1 Software Use Case Diagram by Authors	81
Figure 7.1.2 Subdiagram	82
Figure 7.2 Software State Diagrams by Authors	83
Figure 7.3 ERD	84
Figure 7.4 State Diagram	85
Figure 7.5: UI	88
Figure 10.1 BOM	99

List of Tables

Table 2.1	System Specifications	7
Table 2.2	Component Specifications	7
Table 3.1	Comparison between MCU and SBC	21
Table 3.2	MPU Comparison by Authors	23
Table 3.3	Antenna Types	27
Table 3.4	Touchscreen Options	35
Table 3.5	Battery Types	39
Table 3.6	Battery Models	40
Table 3.7	Solar Panel Types	42
Table 3.8	Solar Models with Features	43
Table 3.9	Charge Controller Models	45
Table 7.1	UI Breakdown	86
Table 10.1	Budget	103
Table 10.2	Table of Work Distributions	105
Table 10.3	SD1 Schedule	105
Table 10.4	SD2 Schedule	106

Chapter 1 Executive Summary

Our project originally began as a passive aircraft detection system, but after changes in scope and limitations in hardware support from our sponsor, we shifted our focus toward developing a **radio diversity system** using low-cost Software-Defined Radios (SDRs) and multiple Raspberry Pi units. The goal of this updated system is to demonstrate that three spatially separated receivers can monitor the same broadcast signal and automatically select the one with the best signal quality in real time. Even though this version of the project is simpler than the original aircraft detection concept, it still shows the core idea behind diversity: using multiple receivers to improve reliability when one receiver experiences noise, fading, or other interference.

In our system, a single FM broadcast is received by three different RTL-SDR units, each connected to its own Raspberry Pi positioned in different locations. These three receivers continuously stream audio and signal-quality metrics to a central Raspberry Pi, which runs our diversity algorithm. The central Pi compares the incoming signals based on strength and clarity, then chooses the best one to output at any given moment. By switching between receivers as conditions change, the system maintains a more stable and clean audio signal than any one receiver could on its own.

To make the system easy to use, we built a touchscreen interface on the central Raspberry Pi. The UI shows live SNR or RSSI values from each receiver, highlights which SDR is currently selected, and includes a spectrum plot to visualize the RF environment. Because all of this runs on a 5-inch HDMI touchscreen, the system can be controlled directly through simple on-screen buttons without needing a keyboard or mouse. The UI also displays backend messages and switching events, giving a clear view of how the diversity logic behaves during tests.

Throughout Senior Design 2, our main focus was on making sure the system was reliable, responsive, and simple enough to operate during demonstrations. We spent time testing each Raspberry Pi individually, confirming real-time audio streaming performance, comparing different network setups, and adjusting switching thresholds. Once each component worked on its own, we tested the full-system setup with all three receivers active at once. Moving the transmitter around or placing obstacles in the environment helped us observe how the signal changed and how quickly the diversity logic reacted. These tests helped fine-tune the behavior of the system until it felt stable and predictable.

Even though this project no longer focuses on detecting aircraft or performing large-scale localization, it still serves as a strong proof-of-concept for a multi-receiver RF system. The design shows how diversity can improve link reliability and demonstrates how multiple low-cost nodes can work together to create a stronger overall signal. The architecture we built can also be expanded in the future—either by using higher-end SDRs, adding more receivers, or re-introducing time-difference-of-arrival methods if the goal shifts back toward location-based detection. For now, the system successfully demonstrates our main objective: real-time signal selection using multiple distributed receivers.

Chapter 2 Project Description

2.1 Motivation and Background

Our project originally started as a passive aircraft detection system, but after changes in scope and hardware availability, we shifted our focus to something more achievable within our Senior Design timeline: building a radio diversity system using multiple RTL-SDR receivers and several Raspberry Pi units. Even though the goal has changed, the overall idea of using multiple RF receivers to improve reliability still remains. Instead of tracking aircraft or decoding ADS-B signals, we now focus on something more fundamental and easier to measure real-time link quality and signal selection.

In many wireless systems, signals do not travel perfectly through the environment. They bounce off walls, get blocked by objects, weaken with distance, and occasionally drop out. Because of this, a single receiver might experience fading or interference that another nearby receiver does not. The concept of diversity takes advantage of this by spreading out multiple receiving antennas and comparing their outputs. If one experiences a moment of fading, another one may still have a clean signal. By switching between them, the system can maintain a more stable connection.

Our project applies this same idea, but in a simple and easy-to-demonstrate way. A single FM broadcast is received at three different locations using RTL-SDR dongles plugged into Raspberry Pi devices. Each Pi demodulates the audio, reads signal strength information, and transmits both the audio and telemetry to a central unit. The central Raspberry Pi compares all incoming streams and picks the one with the best quality at that moment. This method demonstrates the basic concept of radio diversity in a way that is practical, measurable, and visually easy to understand on our touchscreen UI.

This updated project direction allowed us to focus on something hands-on and achievable while still relating to larger RF system concepts. The architecture we built can still be expanded later into multi-node RF monitoring or even back into TDOA-based localization. But for now, our main focus is developing and demonstrating the diversity concept using low-cost hardware.

2.2 Existing Products

Radio diversity is a widely used concept in wireless communications. Many commercial and research systems rely on similar techniques to handle fading, interference, and obstacles in real-world environments.

2.2.1 General Applications

- **Cellular and Wi-Fi Systems:** Modern mobile devices use multiple antennas to improve reception. This helps combat multipath fading and ensures a more stable link indoors or in crowded areas.
- **Satellite Communications:** Ground stations commonly use diversity between multiple receivers to handle weather changes or signal blockage.
- **SDR Research & MIMO Experiments:** Universities and labs frequently use multi-antenna setups to understand propagation, run diversity experiments, or implement beamforming and spatial multiplexing.

2.2.2 Proprietary Applications

- **GDP Space Systems – Best Source Selector:** A high-end commercial product used in aerospace telemetry that automatically selects the strongest RF link from multiple antennas. This is similar to what our system does, but built for mission-critical and high-budget applications.
- **Analog Devices RadioVerse Platforms:** These provide multiple synchronized RF channels and advanced DSP features, but they require specialized hardware and expertise.

2.2.3 Comparison of Our Design to Available Products

When comparing our SDR Diversity System to existing products, the biggest difference is how simple and affordable our setup is compared to the professional equipment used in industry. Most commercial diversity systems are built for very specialized applications like aerospace telemetry or advanced wireless infrastructure, and they usually rely on high-end, phase-coherent RF front ends. These systems are extremely powerful, but they come with a huge price tag and require technical knowledge far beyond what most student teams have access to. Our design, on the other hand, is built entirely from low-cost, plug-and-play hardware such as RTL-SDRs and Raspberry Pis, which makes it much more accessible while still getting the main idea across.

For example, GDP Space Systems has a product called the Best Source Selector, which is used in places like missile test ranges and satellite facilities. It constantly monitors multiple RF links and automatically picks the strongest one, similar to what we are doing. However, their system is built for mission-critical environments and is basically a rack-mount box full of carefully engineered hardware. It's fast, reliable, and incredibly robust—but also very expensive and completely closed off, so you can't modify or learn much from how it works internally. Our setup doesn't try to compete with that level of performance, but it does demonstrate the same core idea in a lightweight way that students can actually build and experiment with.

Another example is Analog Devices' RadioVerse platforms. These are advanced SDR systems with multiple synchronized receive channels that support things like MIMO and beamforming. They are amazing tools for research labs because they let you run complex DSP algorithms and control everything down to the hardware level. But they require detailed RF engineering knowledge and usually need custom PCBs, high-speed data interfaces, and professional-grade tools. Our project takes the opposite approach by sticking to simple USB SDRs and using

software to do the diversity decision-making. This trades performance for accessibility, which makes more sense for a proof-of-concept diversity demo.

What sets our design apart is that it's something almost anyone can replicate without spending thousands of dollars or needing specialized equipment. It uses everyday components like Raspberry Pis, small antennas, and Python scripts, but it still shows real diversity behavior in a way you can actually see and hear during testing. Even though it's not meant for high-end or mission-critical use, our system still provides a good hands-on example of how diversity can improve signal reliability. That's a big part of why our system matters—it bridges the gap between expensive professional solutions and the kind of practical, low-cost system that students and hobbyists can learn from.

2.3.1 Basic Goals

The basic goals represent the essential outcomes needed to prove the feasibility of the project. These deliverables establish the foundation of the system and must be achieved for the design to be considered successful:

- Implement a three-antenna receiver system feeding into a single SDR.
- Demonstrate selection diversity by comparing link quality across branches.
- Use simple DSP metrics (RSSI or SNR) to select the best signal in real time.
- Validate performance improvement over a single-antenna baseline in controlled tests.
- **Demonstrate 3 types of diversity:** Build 3 options for radio diversity, including
 - **Spatial Diversity:** Antennae are in different locations from the SDR
 - **Frequency Diversity:** Different broadcasts on different frequencies
 - **Time Diversity:** Signals are sent in sequence to each antennae

2.3.2 Advanced Goals

Advanced goals go beyond the fundamental proof of concept by adding features that improve system stability, performance, and usability. These goals will help refine the design into a more practical and field-ready solution:

- **Control Methods:** Implement 3 control methods within the system, including:
 - **Threshold Control:** Input is automatically switched to signal with best reception only when the current input drops below given threshold
 - **Automatic Control:** Input is automatically switched to signal with best reception instantaneously
 - **Manual Control:** Any given signal can be manually selected

- **Signal quality metrics:** Add advanced estimators such as Error Vector Magnitude (EVM) or Packet Error Rate (PER).
- **Switching logic improvements:** Implement hysteresis and smoothing to ensure stable selection behavior.
- **Visualization:** Provide a logging and display interface showing per-branch quality, active selection, and switching history.
- **Field-ready integration:** Package the system into a solar-powered, weatherproof enclosure suitable for extended outdoor use.

2.3.3 Stretch Goals

Stretch goals outline ambitious additions that are not required for the core project but would demonstrate scalability and innovation if achieved. These represent opportunities for future research or extended development:

- **TDoA exploration:** Investigate time-difference-of-arrival methods to support aircraft localization.
- **Scaling branches:** Expand the system to more than three receivers to demonstrate modular growth.
- **Machine learning–based selection:** Experiment with predictive models to anticipate the best link based on environmental patterns.
- **Remote telemetry:** Stream diversity metrics and selected outputs over SATCOM for real-time remote monitoring.

2.3.4 Objectives

The objectives provide measurable criteria for project success. Each objective sets a clear benchmark that ensures progress can be tracked and the final system evaluated effectively:

1. **Demonstrate diversity gains** by showing reduced fading/outage duration compared to a single antenna.
2. **Develop real-time DSP algorithms** for link quality estimation and branch switching with <50 ms latency.
3. **Implement stability measures** (hysteresis/smoothing) to reduce unnecessary switching.
4. **Design with COTS hardware** to ensure low cost, modularity, and replicability.

5. **Conduct controlled field tests** (multipath fading, human blockage, interference) to validate system performance.
6. **Ensure scalability** so the architecture can be adapted for TDoA or multi-branch expansion in future research.

2.4 Features and Functionalities

2.4.1 Design Implementations

- **Radio Diversity Detection:** The system employs a single broadcast source and three spatially distributed receivers feeding into a shared Software-Defined Radio (SDR). Advanced DSP algorithms evaluate the incoming streams in real time and automatically select the strongest link, mitigating multipath fading and environmental interference.
- **Digital Signal Processing (DSP):** Real-time DSP techniques are central to the system's operation, enabling dynamic signal quality assessment, adaptive link switching, and robustness to interference or obstruction.
- **COTS-Based SDR Hardware:** The design leverages Commercial Off-The-Shelf (COTS) SDR platforms, ensuring affordability, accessibility, and flexibility for a wide range of users without requiring proprietary or custom RF hardware.
- **Modular and Scalable Architecture:** While the prototype demonstrates three receivers, the design can be expanded to support additional channels, enabling enhanced diversity performance or future migration toward MIMO-style configurations.
- **Ease of Setup:** The system is designed for ease of setup, requiring minimal configuration to begin operation. All receivers and SDR components integrate seamlessly, allowing field deployment in under one hour.
- **Durability:** Using weather-protected enclosures and stable power supply configurations, the system can operate reliably in diverse environments, from urban monitoring setups to remote field deployments.

2.4.2 Customer Input

- **Affordability:** Customers need a low-cost diversity solution for research, border security, and academic exploration. This emphasized the importance of keeping costs low while maintaining functionality, which directly influenced the decision to use Commercial Off-The-Shelf (COTS) components, ensuring ease of access and a unit cost under \$x,xxx.
- **Ease of Use:** Customers requested a system that could demonstrate real-time diversity without extensive technical training. In response, the design prioritizes simplified DSP controls and modular setup, enabling non-specialist users to deploy and operate the system quickly.
- **Reliability in Variable Environments:** Customers highlighted the need for reliable operation in challenging RF conditions such as multipath, signal fading, and interference. This informed the system's strongest-link selection DSP logic, ensuring resilient communications in diverse environments.

2.5 Specifications

2.5.1 System Specs

Table 2.1 System Specifications

Parameter	Specifications	Description
Cost	< \$400 USD	Amazon COTS parts only
Receiver Branches	3 Antennas/Receivers	Minimum for Diversity Benefit
Operating Band	FM 88–108 MHz (test band)	Strong, legal signals for POC
Fading Signal and Switching Latency	< 1s to detect fading signal and < 50ms to switch	Ensures smooth switching
Audio Delay	< 3s	Ensure smooth playback of audio between SDRs
Performance Gain	≥ 6 dB SNR or $\geq 2\times$ error-rate reduction vs single antenna	Demonstrates clear diversity improvement
Power Requirement	≤ 10 W	Runs from USB power bank or 12 V source
Touchscreen Latency	< 500ms	Time between button press and touchscreen response
Selection Accuracy	90%	Accuracy of highest SNR being selected at any given moment

2.5.2 Component Specs

Table 2.2 Component Specifications

Component(s)	Specification	Example/Notes
Antenna 3x	Tuned to test band, omnidirectional	Simple VHF Dipole (Amazon)
Receiver 3x	Low-cost USB SDRs, identical models	RTL-SDR dongles
Host Computer	Quad-core, 2–4 GB RAM,	Raspberry Pi 4 or mini-PC

	USB 3.0	
Power Supply	USB-C PD bank ($\geq 20,000$ mAh)	Portable, low-cost Amazon option
Software	Real-time DSP & selection	GNU Radio / Python scripts

2.6 House of Quality

The House of Quality for the Radio Diversity Automation project clearly maps customer requirements to measurable engineering specifications, facilitating a structured approach to product development. The matrix is centered around aligning five primary customer needs—Affordability, Ease of Use, Real-Time Monitoring, Energy Efficiency, Reliability, and Adaptability to Region—with critical engineering metrics like Cost, Power Consumption, Localization, Time to Detect, Bandwidth, and System Accuracy.

Affordability strongly influences cost, as indicated by the black circle, showing a strong correlation. The goal is to stay below the \$8,000 target, which ensures the system remains accessible for wide-scale deployment, particularly in developing or remote areas. This is supported by using COTS components and open-source software.

Ease of Use relates moderately to system software and detection time. The project targets a plug-and-play design that enables setup in under one hour. The software interface is designed to be intuitive, reducing the learning curve and ensuring non-specialists can operate the system effectively in the field.

Real-Time Monitoring has direct connections to detection software, bandwidth, and system accuracy. The system aims to capture, process, and transmit aircraft signals in near-real-time, leveraging digital signal processing and cloud integration via Starlink. Bandwidth allocation and efficient data compression are optimized to maintain fast and secure updates without overwhelming the system.

Energy Efficiency is tied closely to power consumption and detection software. The system employs solar-powered operation with intelligent sleep cycles and lightweight edge computing to reduce energy draw. This ensures 24/7 functionality in isolated locations where grid power is unavailable.

Adaptability to Region is related to all engineering characteristics to some extent. The design is modular, scalable, and climate-resilient—able to be deployed in forests, deserts, coastlines, or mountainous regions without requiring region-specific redesigns.

At the top of the matrix, correlations between engineering characteristics are explored. For example, improving accuracy might increase cost or power consumption, so trade-offs must be considered carefully during design optimization. Positive and negative correlations are marked to guide these design decisions.

This House of Quality ensures that customer expectations drive technical priorities while allowing for traceability from requirements to design metrics, ensuring that all engineering decisions are justified and aligned with the intended mission.

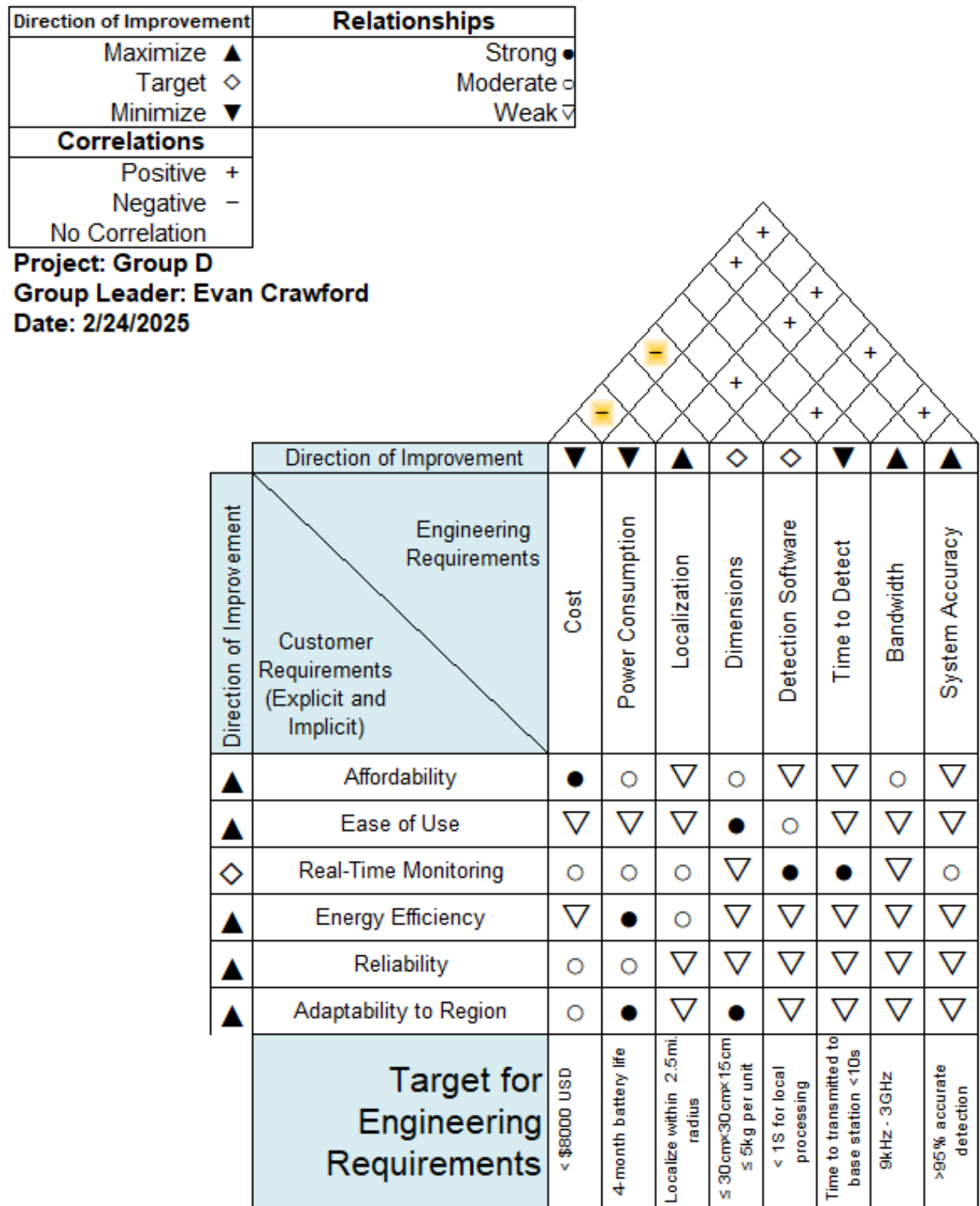


Figure 2.1 House of Quality Diagram

2.7 Block Diagrams

2.7.1 Hardware Block Diagram of Node Array

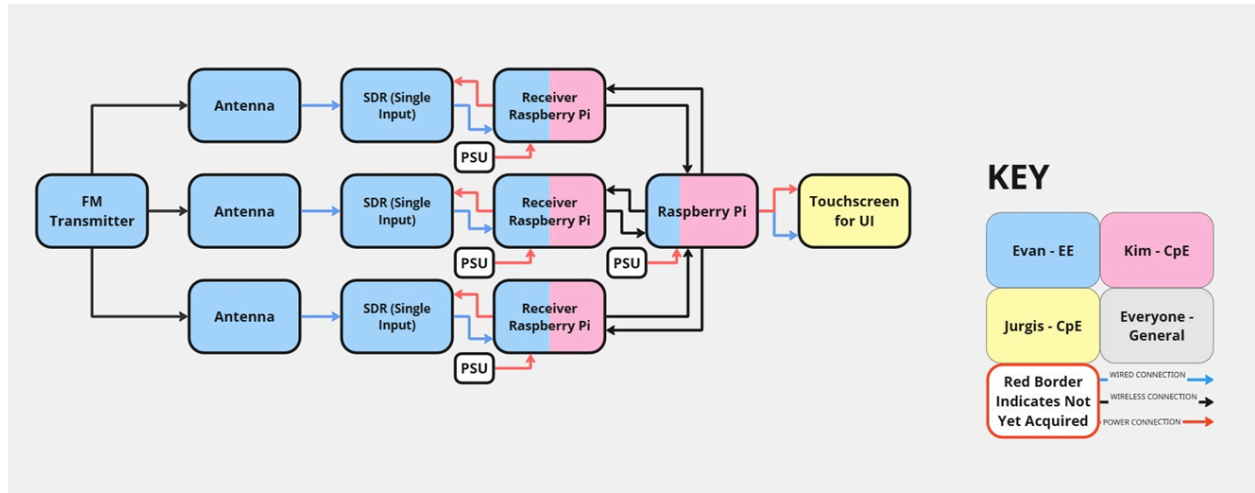


Figure 2.2 Hardware Block Diagram of System

This block diagram shows the setup of our system, which consists of three antennae, the SDR and the raspberry pi, along with some optional filtering.

- The SDR serves as the central hub, aggregating data from all the antennae.
- Three spatially separated antennae feed into the SDR front end, where real-time DSP on the raspberry pi evaluates signal quality and selects the strongest link for reliable monitoring.
- In an additional phase, a laptop or remote terminal could access the processed data for visualization and analysis, but in this case the system operates locally without reliance on external satellite or cellular services.
- The system is powered by a single power supply.

With the radio-diversity design, the system gains resilience against fading and interference by comparing signals from three spatially separated antennas. Each antenna receives the same broadcast, but variations in environment and orientation cause differences in signal strength and quality. The SDR and Raspberry Pi continuously analyze these inputs using DSP, automatically selecting the strongest link in real time. This approach expands on the original concept by maintaining reliable reception under changing conditions while still preserving modularity and long-term operation.

2.7.2 Software Flow Chart

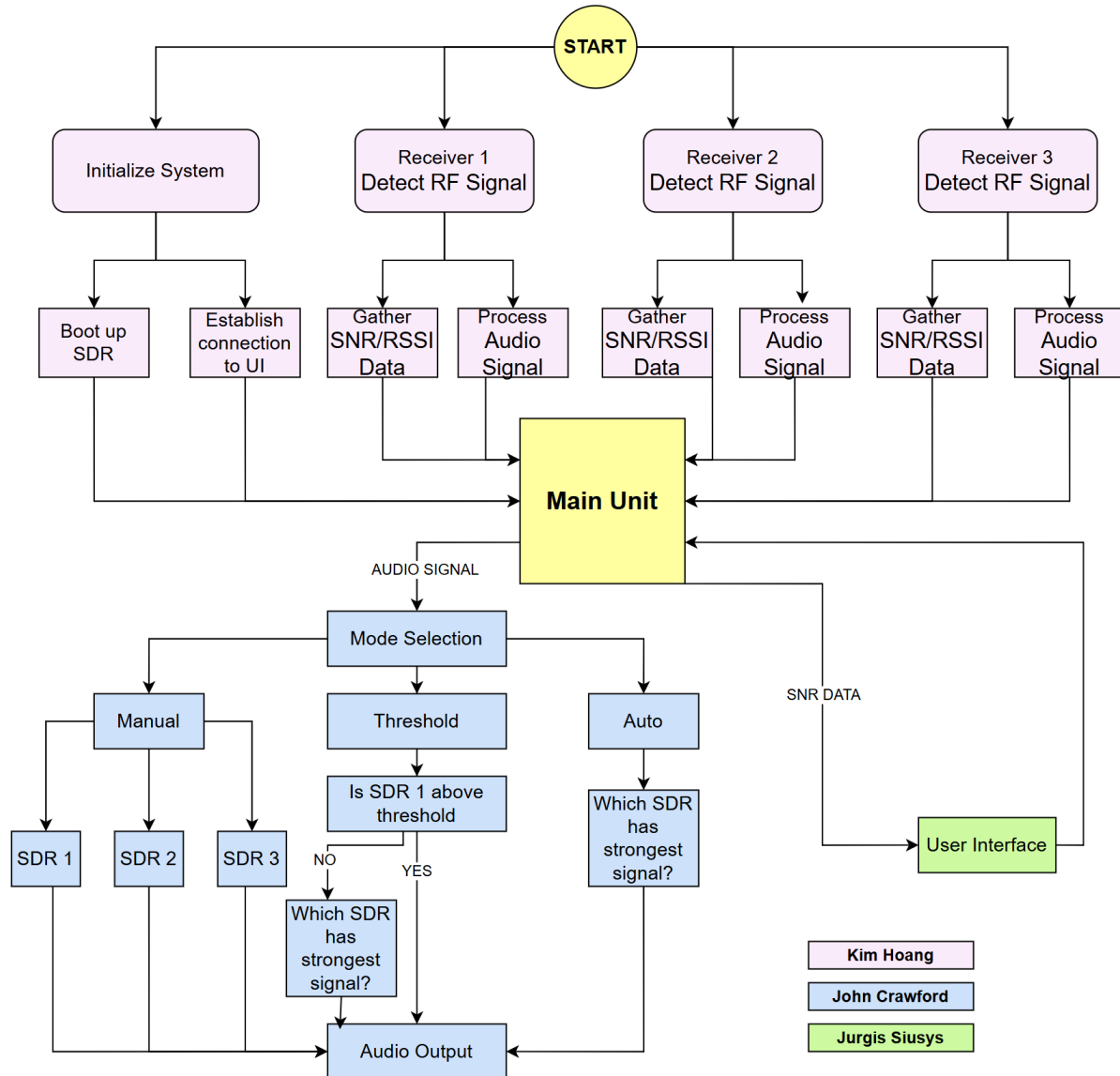


Figure 2.3 Software block diagram

Figure 2.3 illustrates the overall software architecture of the Radio Diversity System, beginning with system initialization and extending through the full signal-processing and decision-making pipeline. When the system starts, the Main Unit initializes all required services, network sockets, and data structures. In parallel, each Software Defined Radio (SDR) receiver powers on and establishes a communication link with both the Main Unit and the User Interface (UI). This includes configuring the RTL-SDR hardware, synchronizing parameters, and preparing TCP channels used for telemetry and audio transport.

Once the receivers are active, each SDR independently detects the incoming RF signal and immediately begins processing the baseband stream. Each receiver performs two essential tasks:

(1) extracting real-time SNR/RSSI telemetry, which provides a measure of link quality, and
 (2) demodulating the RF signal into audio samples, which are sent to the Main Unit as timestamped audio frames.

This information flows continuously from all three receivers to the Main Unit, allowing parallel processing and high-resolution monitoring of signal conditions.

At the core of the architecture is the Main Unit, which aggregates the audio and telemetry feeds from all receivers. The Main Unit is responsible for executing the diversity selection logic, which determines which SDR provides the audio output at any moment. Three diversity modes are supported:

1. Manual Mode – The operator explicitly chooses SDR 1, SDR 2, or SDR 3. The Main Unit outputs audio exclusively from the selected receiver, independent of signal quality. This mode allows direct operator control for troubleshooting, testing, and evaluating individual SDR performance.
2. Threshold Mode – The Main Unit first evaluates whether SDR 1 meets a predefined SNR threshold.
 - If SDR 1 satisfies the requirement, its audio stream is selected.
 - If SDR 1 fails to meet the threshold, the Main Unit automatically compares the SNR across all receivers and selects the SDR with the strongest signal.
 This mode provides a hybrid decision mechanism that prioritizes SDR 1 while retaining automatic fallback behavior in degraded conditions.
3. Auto Mode – The Main Unit continuously compares SNR measurements from all receivers and automatically selects the SDR with the highest signal quality. This mode provides fully automated radio diversity and seamless switching based solely on real-time telemetry.

The selected audio stream is then forwarded to the Audio Output subsystem for playback. Simultaneously, the Main Unit transmits updated telemetry—including SNR values, connection status, and the currently focused SDR—to the User Interface, where operators can visualize system performance and monitor receiver activity in real time.

Overall, the flowchart captures the complete operational pipeline of the system, emphasizing the distributed SDR architecture, the centralized diversity decision engine, and the real-time feedback provided to the user through the interface.

Chapter 3 Research and Investigation

3.1 Hardware Technologies

3.1.1 Central Processor

3.1.1.1 MPU vs. SBC

3.1.1.1.1. Pros and Cons of SBCs

Single Board Computers (SBCs) are compact, all-in-one computing devices that integrate a processor, memory, storage, and input/output interfaces onto a single board. They offer a cost-effective and energy-efficient solution for various applications, including embedded systems, IoT devices, and edge computing. SBCs are highly versatile and can run full operating systems like Linux, Android, or Windows, allowing them to support a wide range of applications. They provide higher processing power than microcontrollers (MCUs), making them suitable for tasks like real-time data analysis, multimedia processing, and AI applications. SBCs also feature multiple connectivity options, including Wi-Fi, Bluetooth, Ethernet, and USB, enabling seamless integration with other devices and cloud services. Additionally, their compact size and low power consumption make them ideal for portable and remote applications.

Despite their advantages, SBCs have some limitations. They typically consume more power than MCUs, making them less suitable for ultra-low-power applications. Unlike traditional computers, they often have limited upgradeability, meaning users are restricted to the built-in CPU and memory. Additionally, SBCs require an external power source and storage (such as an SD card or eMMC), which can introduce additional failure points. For real-time processing applications, SBCs may lack the low-latency performance required compared to dedicated MCUs or FPGAs. Finally, while SBCs are more affordable than full-fledged computers, higher-end models can be costly, especially when additional components like cooling systems or external peripherals are needed.

3.1.1.1.2.Pros and Cons of MCUs

MCUs are highly integrated and self-contained, meaning they include a CPU, RAM, Flash memory, and I/O peripherals on a single chip. This integration makes them compact, power-efficient, and cost-effective, which is ideal for applications that need to run autonomously in remote locations. They consume very little power, making them perfect for battery-powered or solar-powered systems, like our aircraft detection project. Additionally, many modern MCUs come with built-in DSP (Digital Signal Processing) capabilities, allowing them to handle signal processing tasks efficiently without needing an MPU.

However, MCUs have limited processing power compared to MPUs. They typically run at lower clock speeds (tens to hundreds of MHz) and have limited memory, which can be a drawback for projects that require complex data analysis or multitasking. Additionally, because MCUs run bare-metal code or lightweight RTOS (Real-Time Operating Systems), they lack support for full operating systems like Linux or Android, which MPUs can run.

3.1.1.1.3.Why SBC could be a best fit?

After evaluating the requirements of our radio diversity, our group decided that a Single Board Computer (SBC) would be the best fit for our project. While Microcontrollers (MCUs) offer ultra-low power consumption and real-time processing, they lack the computational power, multitasking capabilities, and extensive connectivity options required for advanced signal processing, data fusion, and cloud integration. Since our system needs to handle complex RF

signal analysis, transmit data over networks, and support potential AI-based enhancements, an SBC provides the necessary balance of power and flexibility without excessive energy consumption.

One of the key reasons for our choice is that SBCs support full operating systems, allowing us to utilize multithreading, advanced networking protocols, and high-speed data storage. Unlike MCUs, which have limited memory and processing power, SBCs like the Raspberry Pi 4, Raspberry Pi Zero W, or similar models enable real-time signal filtering, data processing, and seamless communication with cloud servers. The availability of built-in Wi-Fi, Ethernet, and USB interfaces ensures reliable data transmission, which is crucial for remote deployment where network stability may vary.

Our group believes that SBCs strike the right balance between processing power, energy efficiency, and ease of integration. They allow for scalability, meaning that as our detection system evolves, we can incorporate machine learning algorithms, automated data analysis, and enhanced visualization tools without the need for extensive hardware changes.

3.1.1.1.4.Choosing between MCU and SBC

Developing a low-cost, radio diversity-based passive RF signal monitoring system for remote environments requires choosing the right processing unit to balance cost-efficiency, signal quality optimization, and real-time decision-making performance. The system’s core task is no longer simply detecting aircraft signals, but rather comparing signal quality across spatially distributed receivers and dynamically selecting the strongest, cleanest link in real time. This introduces computational demands that go beyond basic signal detection—requiring fast feature extraction (SNR, RSSI, BER), diversity path analysis, secure data packaging, and uplink via SATCOM. While Microcontroller Units (MCUs) are ideal for ultra-low-power, single-purpose applications, they generally lack the memory and processing headroom to perform advanced digital signal processing, manage encrypted communication pipelines, or handle diversity-based link optimization at scale. In contrast, Single-Board Computers (SBCs) or Microprocessor Units (MPUs) offer the flexibility, I/O versatility, and compute power necessary for multi-threaded processing, real-time data comparison across multiple units, and integration with cloud services. The table below compares key features of MCUs and SBCs in the context of our system needs—highlighting why an SBC is the optimal platform for implementing a radio diversity framework that prioritizes resilience, adaptability, and field-deployable intelligence.

Table 3.1 Comparison between MCU and SBC

Feature	<i>MCU ((Microcontroller Unit)</i>	<i>SBC (Single Board Computer</i>
Integration	Includes CPU, RAM, Flash storage, I/O peripherals on a single chip.	A complete computing system on a single board with CPU, RAM, storage, and I/O controllers.

Processing Power	Low to Medium (Tens to hundreds of MHz). Optimized for real-time processing	High (Above 1 GHz). Designed for multitasking and running complex applications.
Power Consumption	Very low (mW to a few watts). Optimized for battery-powered applications.	Higher (W to tens of watts). Requires a stable power supply and may need cooling.
Operating System	No OS or runs Real-Time Operating System (RTOS) like FreeRTOS, Zephyr.	Supports full OS like Linux, Android, Windows Embedded.
Programming Languages	Primarily C, C++, Assembly, and MicroPython for real-time control.	Supports Python, Java, C++, Go, Linux-based tools for high-level computing.
Memory & Storage	Fixed, built-in RAM and Flash storage (e.g., 64KB-512KB RAM, 256KB-2MB Flash).	Uses external RAM and storage (e.g., DDR4 RAM, eMMC, SD card, or SSD).
Communication Protocols	I ² C, SPI, UART, CAN (lightweight, low-power communication).	Supports high-speed data transfer and networking like Ethernet, Wi-Fi, USB, PCIe, and MIPI.
Use Case	Best for IoT, automation, motor control, medical devices, and low-power embedded systems.	Best for industrial automation, AI, multimedia, networking, and complex computing.
Real-time Processing	Designed for real-time applications, ensuring low latency.	Not optimized for real-time tasks, but supports multithreading and high-level computation.
Multitasking	Limited to simple control loops and task switching.	Supports multitasking, runs multiple applications simultaneously.
Peripheral Control	Optimized for direct hardware control (e.g., sensors, motors, displays)	Handles complex peripherals like cameras, touchscreens, GPUs, and multimedia interfaces.
Ease of Use	Easier to program and deploy with simple hardware requirements.	More complex, requires OS configuration and software drivers.
Cost	Low cost (\$1-\$15 per unit),	Higher cost (\$20-\$200 per

	minimal external components needed.	unit), requires additional components.
Example Devices	Arduino (ATmega328), ESP32, STM32, PIC16F, MSP430.	Raspberry Pi (Broadcom), Qualcomm Snapdragon, Intel Atom, NXP i.MX.
Power Supply Requirement	Can run on low power (coin-cell batteries, USB, small power supplies).	Requires high power (dedicated power supply, battery packs, high-current sources).

After evaluating the technical requirements of our radio diversity-based passive aircraft signal monitoring system, we have selected a Single Board Computer (SBC) over a Microcontroller Unit (MCU) as the primary processing platform. While MCUs offer low power consumption and are effective for basic real-time control tasks, they fall short in handling the complex signal processing, inter-unit communication, and multi-source comparison logic required in a radio diversity architecture. Our system continuously receives RF signals from three spatially distributed receivers, performs real-time signal quality analysis (e.g., SNR, RSSI, BER), and dynamically selects the strongest path—all of which demand a processor with higher throughput and multitasking capabilities. SBCs like the Raspberry Pi 4 or Raspberry Pi Zero W offer the necessary processing power, memory capacity, and Linux OS support to manage simultaneous data reception, signal feature extraction, secure encryption, and satellite communication via Starlink. Furthermore, their compatibility with cloud services and AI tools positions them well for future upgrades, including machine learning-based signal classification and TDoA localization. With scalability, adaptability, and field-deployable robustness in mind, an SBC is the more suitable choice for a distributed, real-time, and computationally intensive passive RF detection system operating in remote, infrastructure-limited environments.

3.1.1.2 SBC Selection

3.1.1.2.1. Raspberry Pi Series

With our project's focus now shifted entirely toward demonstrating radio diversity and signal quality optimization—and in the absence of external sponsorship—cost-effective yet capable computing platforms have become even more important. Our goal is to implement a system that compares RF signal quality across multiple spatially separated receivers, dynamically selects the best link in real time, and securely transmits the data via Starlink SATCOM. To achieve this, we carefully evaluated several models from the Raspberry Pi series, comparing their suitability for RF signal processing, inter-unit communication, and reliable performance in remote deployment scenarios.

Each Raspberry Pi model brings unique strengths in terms of processing power, connectivity options, power efficiency, and scalability. The platform's evolution from the early Model 1 to the advanced Raspberry Pi 4 has made it an ideal candidate for lightweight embedded systems,

especially for distributed projects like ours that require multiple computing nodes to operate collaboratively under the radio diversity paradigm.

The Raspberry Pi Model 1 introduced basic computing for education and DIY applications, with a single-core 700 MHz ARM11 processor and 256MB to 512MB RAM. Though revolutionary at the time, its limited processing power and lack of modern connectivity features make it unsuitable for the multi-threaded DSP tasks and packet encryption required in our system.

The Raspberry Pi 2 improved significantly with a quad-core ARM Cortex-A7 processor (900 MHz) and 1GB of RAM. It retained the GPIO and I/O flexibility of earlier versions while offering more stability for multitasking. Although still modest by today's standards, this model laid the groundwork for distributed IoT systems, making it theoretically viable for small-scale RF processing, albeit with constraints on processing speed and memory bandwidth.

The Raspberry Pi 3 further enhanced the platform by integrating built-in Wi-Fi and Bluetooth, eliminating the need for external communication modules—an essential feature for our radio diversity setup, which relies on each node sharing signal metrics with a centralized decision unit. The 1.2 GHz quad-core ARM Cortex-A53 processor and 1GB of RAM enabled more robust signal processing and inter-node communication. These capabilities made it a practical choice for low-cost deployments where basic diversity comparison and remote communication were needed without significant performance overhead.

The Raspberry Pi 4 Model B, however, is the standout choice for our system. It features a 1.5 GHz quad-core ARM Cortex-A72 CPU, and comes in RAM configurations of 2GB, 4GB, or 8GB. These specs provide the necessary headroom for real-time digital signal processing, multi-source quality comparison, and encryption of data for Starlink uplink transmission. With dual-band Wi-Fi, USB 3.0, Gigabit Ethernet, and dual HDMI ports, the Pi 4 supports high-speed communication between units, the main controller, and external services. Its improved thermal efficiency and support for large-scale Linux-based applications make it the optimal choice for our distributed signal quality selection and transmission pipeline.

In our radio diversity-based system, each Raspberry Pi acts as an autonomous RF monitoring node. It receives analog signals through connected RF front-ends, performs local signal quality analysis (measuring SNR, RSSI, and BER), and transmits its results to the Main Unit for diversity comparison. This requires both local computational capability and high-speed networking support—tasks for which an MCU would be underpowered and inefficient. Furthermore, the Pi's ability to run Linux enables us to build a robust and flexible software stack using familiar tools like Python, C++, and shell scripting, while also allowing future integration with machine learning models or time-difference-of-arrival (TDoA) calculations.

Unlike traditional embedded MCUs, Raspberry Pi boards provide sufficient processing power to handle multi-threaded DSP, perform live signal scoring, and initiate encrypted Starlink

communication. Their affordability makes them a sustainable choice for multi-node deployment, especially in the absence of sponsorship. Even in the most constrained environments, a network of Raspberry Pi-based receivers can be deployed, powered by solar panels, and maintained with minimal cost.

By leveraging the Raspberry Pi’s capabilities for signal acquisition, diversity comparison, and cloud transmission, we ensure our system remains robust, real-time, and optimized for remote deployment—even without the support of external funding. This strategy not only demonstrates the power of radio diversity in passive RF systems but also sets the stage for future expansions, including AI-enhanced signal classification and full 3D aircraft localization using scalable node arrays.

3.1.1.2.2 Raspberry Pi Zero (W)

As we pivot toward a radio diversity-based architecture and operate without sponsorship, the Raspberry Pi Zero emerges as a highly affordable and practical choice for lightweight detection nodes. Powered by a Broadcom BCM2835 processor (1 GHz) and equipped with 512MB RAM, the Pi Zero offers low power consumption and a compact form factor—both of which are ideal for remote RF signal monitoring in areas with limited infrastructure or solar-only power sources. Though modest in computing capabilities, it is well-suited for collecting basic RF metrics like SNR, RSSI, and BER from localized receivers and forwarding them to a central processing unit for diversity evaluation.

The Raspberry Pi Zero W enhances this utility by integrating built-in Wi-Fi and Bluetooth, enabling seamless wireless communication between nodes and the central decision-making unit (e.g., a Pi 4). This eliminates the need for wired networking, allowing easy deployment in difficult terrain or isolated areas where cabling is impractical. In our radio diversity setup, Pi Zero W boards can serve as secondary or edge nodes—gathering RF signal quality data from spatially distributed receivers, performing minimal pre-processing, and relaying results to the Main Unit responsible for link comparison and selection.

While the Pi Zero lacks the computational resources to run complex AI models or full digital signal processing chains, it plays a critical role as a low-cost, low-power signal capture device in a distributed sensor network. Its simplicity, affordability, and wireless capabilities make it an excellent option for expanding the system’s spatial coverage and ensuring that the diversity logic is fed with geographically varied signal inputs.

3.1.1.2.3 Raspberry Pi Compute Module Series

The Raspberry Pi Compute Module (CM) series offers a compelling solution for building compact, modular, and industrial-grade nodes in a radio diversity signal selection system, especially when space constraints or customized I/O requirements are a concern. Designed for embedded applications, the Compute Modules strip away standard ports like HDMI, USB, and Ethernet, exposing only the core SoC and memory through board-to-board connectors. This allows for fully custom hardware designs via carrier boards—making it ideal for precision-tuned

RF receiver arrays, long-term installations, or ruggedized deployments where reliability is critical.

The Compute Module 4 (CM4), based on the same Broadcom BCM2711 processor as the Raspberry Pi 4, delivers a powerful quad-core Cortex-A72 CPU with up to 8GB RAM, optional eMMC storage, and PCIe connectivity for high-speed data transfer. In a radio diversity system, a CM4-based main unit could serve as the central controller—capable of aggregating signal data from multiple distributed Zero W nodes, executing DSP comparisons, and transmitting selected high-quality signals via Starlink SATCOM.

For more constrained processing tasks, the Compute Module 3 (CM3)—derived from the Raspberry Pi 3 architecture—offers a balance of moderate computing capability and low power draw. It is well-suited for mid-tier roles such as regional signal preprocessors, especially in larger deployments where multiple diversity sub-zones need to be monitored and passed upstream.

One of the biggest strengths of the Compute Module platform lies in its hardware flexibility and long-term deployment potential. Developers can tailor each unit's carrier board to fit specific deployment environments—whether that means adding industrial RF filters, rugged power management systems, or heat dissipation mechanisms. This modularity ensures that the system is not only cost-effective and scalable, but also durable enough for real-world field conditions where environmental resilience is required.

In the context of our sponsor-less, radio diversity-focused project, the Compute Module series provides the ability to build customized, high-performance diversity processing units without the overhead of unnecessary hardware. Whether used for the central signal comparison unit or integrated into custom RF hubs, these modules offer the right balance of power, flexibility, and professional-grade reliability—essential for building a scalable aircraft detection system from COTS components.

3.1.1.2.4. Qualcomm Snapdragon

The Qualcomm Snapdragon series is known for its high-performance computing, AI acceleration, and advanced wireless capabilities, making it a strong choice for mobile AI, industrial edge computing, and high-throughput IoT systems. However, in the context of our radio diversity-based RF signal comparison system, Snapdragon platforms are not an optimal fit due to their power complexity, development overhead, and cost—all of which conflict with our project's emphasis on simplicity, efficiency, and affordability, especially after the loss of external sponsorship.

Snapdragon processors such as the 8cx, 865, 888, and 820E offer advanced multicore CPUs, integrated GPUs, and support for mobile AI acceleration, but they are primarily designed for high-performance applications like smartphones, AR/VR, and real-time video processing. These use cases demand significant power and require specialized power management, external memory interfaces, and complex PCB layouts. In contrast, our system focuses on basic RF signal capture and analysis across spatially distributed nodes, where simplicity, low power consumption, and easy integration are more valuable than raw processing power.

For our radio diversity setup, where each node is responsible for detecting and reporting signal quality metrics (e.g., SNR, RSSI, BER) to a centralized main unit for link comparison and selection, Snapdragon MPUs introduce unnecessary hardware and software complexity. Unlike Single Board Computers (SBCs) such as the Raspberry Pi 4, Snapdragon-based solutions often require custom carrier boards, proprietary development environments, and additional middleware, which increase development time and cost. Additionally, the AI and graphics capabilities provided by Snapdragon's DSPs and GPUs are excessive for a system that does not rely on visual processing or advanced AI inference at the edge.

From a cost-efficiency and scalability perspective, Snapdragon systems are better suited for data-intensive edge applications with high bandwidth and processing needs. In contrast, our project requires a distributed, low-power, real-time signal analysis system that can be deployed across multiple remote receivers, possibly running on solar power. Using SBCs like the Raspberry Pi 4 or Pi Zero W allows us to maintain software portability, use open-source tools, and integrate quickly with RF front-ends and Starlink SATCOM—all without exceeding budget or design constraints.

In conclusion, while Qualcomm Snapdragon MPUs provide impressive capabilities for high-end embedded systems, they do not align with the lightweight, scalable, and energy-efficient architecture required for our radio diversity implementation. Prioritizing practical deployment, system simplicity, and total cost of ownership, SBCs remain the best choice for our passive RF signal quality comparison system.

3.1.1.2.5. NXP i.MX

The NXP i.MX series includes a wide range of high-performance MPUs designed for industrial automation, embedded systems, and edge AI applications. These processors offer impressive scalability, secure boot support, and power-optimized cores—making them well-suited for complex use cases like multimedia interfaces, AI inference, and high-speed sensor fusion. However, for our radio diversity-based RF signal quality comparison system, the i.MX platform introduces unnecessary complexity, power consumption, and hardware design overhead, making it less practical given our current constraints and objectives—especially in the absence of external sponsorship.

For instance, the i.MX 6 series, with its integrated GPU and multimedia support, is tailored for applications requiring video rendering or advanced graphical interfaces. These features are excessive for our system, which performs lightweight signal metric extraction (e.g., SNR, RSSI, BER) from passive RF data and transmits selected results via Starlink SATCOM. Similarly, while the i.MX 7 series is designed with IoT in mind and offers dual-core architecture optimized for power efficiency, it still requires custom PCB development, external memory integration, and specialized software environments, which introduce unnecessary barriers compared to readily deployable SBC platforms.

The i.MX 8 series, equipped with AI-ready cores, PCIe, USB 3.0, and multi-display output, is intended for compute-heavy edge systems and real-time analytics pipelines. However, in a distributed, solar-powered radio diversity system—where each node passively listens, performs basic signal quality checks, and sends compact metrics to a central unit—such performance is

largely underutilized. Implementing an i.MX solution would increase power draw, demand more complex power regulation, and extend development timelines—outweighing any benefits in this use case.

Given our project's focus on scalability, ease of deployment, and energy efficiency across remote RF monitoring nodes, the NXP i.MX series is simply too feature-rich and hardware-intensive for our needs. Single Board Computers (SBCs) like the Raspberry Pi 4 or Pi Zero W provide a far more balanced solution: they offer sufficient processing power, built-in wireless connectivity, Linux OS compatibility, and community-supported tools—all within a lower-cost, plug-and-play form factor ideal for rapidly deploying distributed signal quality sensing units.

In summary, while the NXP i.MX platform excels in industrial and high-performance embedded applications, it does not align with our radio diversity implementation goals, which prioritize low-cost, low-power, and quick-to-deploy systems that can function reliably in remote environments without the support of extensive hardware customization or complex development frameworks.

3.1.1.2.6. Intel Atom, Core, and Xeon

The Intel Atom, Core, and Xeon processor families offer outstanding performance for industrial automation, edge AI, and data center applications. However, these platforms are not appropriate for our radio diversity-based RF signal comparison system, which prioritizes low power consumption, remote deployability, and simplified integration—especially in light of our project's transition to a self-funded, cost-conscious model.

The Intel Atom E3900 series is designed for industrial IoT and edge analytics, providing multimedia support, real-time processing, and hardware-level security. While well-suited for smart city and autonomous control applications, Atom-based platforms often lack integrated wireless networking and require external memory, storage, and complex power regulation, all of which increase hardware complexity and deployment costs. In contrast, for our use case—capturing lightweight RF signal metrics (SNR, RSSI, BER) and forwarding them for centralized diversity evaluation—this level of hardware customization is impractical and unnecessary.

Similarly, Intel Core processors (i3 through i9) are powerful solutions for desktop, workstation, and mobile computing. These processors offer multi-core execution, high clock speeds, and advanced instruction sets, but they are overkill for a distributed RF monitoring system intended to run on solar-powered or battery-backed nodes. Their need for external RAM, cooling solutions, and high-current power supplies makes them unfit for field deployments where low energy consumption and compact design are essential.

The Intel Xeon D and Xeon Scalable processors are even more advanced, optimized for enterprise-level tasks such as AI training, cloud orchestration, and large-scale predictive analytics. While impressive, their server-grade power requirements, infrastructure complexity, and high cost place them far outside the scope of our project, which emphasizes simplicity, remote operability, and deployment scalability without specialized data center resources.

Although these Intel platforms offer top-tier computational capabilities, they do not align with our project’s goals of affordability, energy efficiency, and ease of deployment for radio diversity-based signal monitoring. A Single Board Computer (SBC) like the Raspberry Pi 4 offers a far more balanced solution—providing sufficient performance for signal processing and Starlink uplink, built-in wireless connectivity, and a compact form factor well-suited for distributed deployment.

3.1.1.2.7. Texas Instruments Sitara

The Texas Instruments Sitara MPU series is engineered for real-time industrial applications, AI-enhanced automation, and embedded control systems. While these processors offer robust features such as multi-core ARM architecture, secure boot, and real-time communication protocols, they are not the most practical choice for our radio diversity-based RF signal evaluation system—especially under current cost and power constraints.

The Sitara AM335x, based on the ARM Cortex-A8, is widely used in motor control and industrial networking. However, it lacks built-in wireless connectivity, which is a critical requirement in our distributed architecture where RF monitoring nodes must communicate wirelessly with the Main Unit. Its limited performance and additional hardware integration requirements make it less attractive than turnkey SBC options.

The AM437x series, with its 1.5 GHz Cortex-A9 core and secure boot capabilities, does offer higher processing performance. Yet, like the AM335x, it is designed for tightly controlled industrial environments and multimedia interfaces—neither of which align with our need for lightweight, solar-powered field nodes that perform simple RF metric extraction and transmit results for diversity analysis.

The AM57x series, equipped with quad-core Cortex-A15 processors and integrated DSPs, is targeted at high-end automation, AI workloads, and real-time analytics. These capabilities exceed what is necessary for our project, which focuses on measuring and comparing RF signal quality across spatially distributed receivers. For such tasks, the Sitara’s hardware acceleration features and advanced machine learning capabilities are unnecessary and overly power-hungry.

Although the Sitara line excels in industrial reliability and secure networking, it demands custom carrier boards, external memory, and advanced development workflows. These requirements are ill-suited for our goal of rapid, low-cost deployment of modular detection units. In contrast, the Raspberry Pi 4 and Pi Zero W offer out-of-the-box functionality, built-in wireless networking, and a large open-source ecosystem—all while keeping costs and integration complexity low.

In summary, while TI Sitara MPUs are powerful and flexible, they do not align with our project’s need for energy-efficient, scalable, and budget-conscious hardware. The Raspberry Pi platform remains a better fit for our radio diversity implementation, ensuring a smooth balance between performance, integration ease, and remote operability in unsupervised environments.

3.1.1.2.8. SBC Comparison

Table 3.2 MPU Comparison by Authors

Feature	SBC (Raspberry Pi 4)	Qualcomm Snapdragon	NXP i.MX	Intel (Atom, Core, Xeon)	TI Sitara
Processing Power	Quad-core Cortex-A72 (1.5 GHz), suitable for lightweight AI and real-time processing	High-performance Kryo CPUs, AI acceleration, optimized for mobile computing.	Scalable from Cortex-A7 to Cortex-A72, good for industrial automation.	Varies: Atom (low-power), Core (high-performance), Xeon (enterprise-grade).	Ranges from Cortex-A8 (AM335x) to Cortex-A15 (AM57x) with DSP integration.
Power Consumption	Low (5V, ~3-5W), optimized for embedded and remote applications.	Moderate (optimized for mobile but still higher than SBCs).	Moderate to high, varies by series, some optimized for low-power IoT.	High (Core/Xeon consume significant power; Atom is more efficient but still higher than SBCs).	Moderate to high, varies by series, designed for industrial use.
Ease of Integration	Plug-and-play, pre-configured OS support (Linux, Raspbian).	Requires custom PCB design and external components	Needs external RAM, storage, and power management, increasing complexity.	Requires external power, RAM, and cooling, complex setup.	Requires custom hardware, industrial-specific configurations.
Connectivity	Built-in Wi-Fi, Bluetooth, Gigabit Ethernet, USB 3.0, dual HDMI.	Advanced 5G, Wi-Fi 6, Bluetooth, but requires additional integration.	Industrial communication protocols (CAN, SPI, I ² C, PCIe, Gigabit Ethernet).	High-speed connectivity with PCIe, Thunderbolt, Ethernet, but bulky.	Supports industrial networking (EtherCAT, CAN, PCIe, Gigabit Ethernet).
Real-Time Processing	Can handle lightweight real-time	Optimized for multimedia	Some models include real-time	Xeon supports HPC and AI	Integrated DSP (AM57x) and

	tasks with optimized programming but lacks dedicated real-time cores.	and mobile AI, lacks dedicated RTOS support.	cores (Cortex-M integration).	inference, but not optimized for embedded real-time systems.	PRU-ICSS enable industrial real-time control.
Software Support	Linux-based OS, supports Python, C++, AI frameworks like TensorFlow Lite.	Supports Android and Linux, but custom software development required.	Supports Linux, FreeRTOS, and Yocto, but needs customization .	Runs Windows, Linux, and AI frameworks, but overkill for embedded systems.	Runs Linux, TI-RTOS, and industrial frameworks, optimized for automation.
Cost	Affordable (\$35-\$100), includes all essential components.	Expensive (\$200+ for Snapdragon-based dev kits).	Mid-range (\$50-\$200), but needs additional components.	High (\$150-\$1000) , depending on model.	High (\$100-\$500+) , industrial-focused pricing.
Use Case in Our Project	Best fit for cost-effective , scalable, real-time RF signal processing.	Overpowered , designed for AI-driven mobile and high-speed computing.	Good for industrial automation, but adds complexity and cost.	Too complex and power-hungry for our lightweight, remote system.	Industrial-grade real-time control, but not necessary for our needs.

In conclusion, after comparing SBCs, Qualcomm Snapdragon, NXP i.MX, Intel Atom/Core/Xeon, and TI Sitara, we found that SBCs like the Raspberry Pi 3 provide the best balance of performance, power efficiency, ease of integration, and cost-effectiveness for our radio diversity project. While MPUs offer higher processing power and industrial-grade features, they come with greater complexity, higher costs, and increased power consumption, making them less suitable for our lightweight, remote deployment needs. The Raspberry Pi 3 ensures seamless real-time processing, built-in networking, and simple software compatibility, making it the most practical and scalable solution for our project.

3.1.1.3 SBC Selection

For our project, we require an SBC that balances processing power, scalability, energy efficiency, and cost-effectiveness. After evaluating various options, we have chosen the Raspberry Pi series, as the best fit for our project. Raspberry Pi SBCs offer a versatile platform with strong community support, a low-cost structure, and sufficient processing capabilities for real-time signal processing and cloud-based data transmission.

Specifically, models like the Raspberry Pi 4 (Broadcom BCM2711) and Raspberry Pi Compute Module 4 (CM4) provide quad-core ARM Cortex-A72 processors, delivering the necessary computational power for RF signal analysis, data fusion, and AI-driven detection enhancements. Unlike traditional MPUs, which require external memory and additional peripherals, Raspberry Pi integrates storage, networking, and GPIO support, making it an ideal choice for embedded applications and IoT-based deployments. Additionally, Raspberry Pi's compatibility with Linux-based operating systems enables us to leverage existing software tools, machine learning frameworks, and real-time data processing applications with minimal customization.

Another critical factor in our decision is expandability and connectivity. The Raspberry Pi 4 and Compute Module 4 support high-speed USB, PCIe interfaces, Gigabit Ethernet, and Wi-Fi 5, ensuring seamless integration with remote monitoring systems, cloud services, and external sensors.

By utilizing Raspberry Pi as our SBC, we ensure an efficient, cost-effective, and scalable solution that meets the real-time processing, and remote deployment needs of our radio diversity project.

3.1.2. Antenna

The antenna is a core component in the success of our SDR-based radio diversity system, as it is responsible for capturing RF signals across multiple bands of interest. To ensure reliable and effective operation, antennas must meet criteria such as wide frequency coverage, stable impedance matching, and durability for long-term outdoor deployment. In a radio diversity system, multiple antenna types can be combined to maximize coverage, mitigate fading, and improve system robustness. Cost, ease of deployment, and scalability also factor heavily into antenna selection, since the system is designed for distributed, modular operation.

Several antenna types may be suited for this design, including Log-Periodic Dipole Arrays (LPDA), Yagi-Uda antennas, dipole antennas, phased arrays, monopoles, and microstrip patches. Each offers unique trade-offs in terms of frequency range, gain, directivity, and deployment complexity. Selecting the optimal set of antennas is essential to ensure high-quality reception, improve diversity performance, and maintain affordability for scaling across multiple nodes.

3.1.2.1 Choosing the Antenna

3.1.2.1.1. Log-Periodic Dipole Array (LPDA) Antennas

LPDA antennas are a strong candidate for this system due to their broadband reception capability, often spanning from VHF into GHz frequencies. This makes them versatile for capturing diverse communication and navigation signals without requiring multiple antennas for different bands. Their directional gain improves sensitivity and helps reduce interference from unwanted directions, a useful trait for distributed diversity nodes.

Limitations include their relatively large size, which may complicate installation in compact or concealed setups. Careful alignment is also necessary to maximize performance. Still, their combination of bandwidth, directionality, and affordability makes them an excellent option for covering wideband signals in diversity systems.

3.1.2.1.2. Yagi-Uda Antenna

The Yagi-Uda antenna is valued for its high gain and narrow beamwidth, making it ideal when long-range, focused reception is required. In a diversity system, Yagi antennas can complement wider-coverage options by offering precision targeting of weaker or distant signals. Their lightweight and relatively simple construction also make them easy to deploy on multiple nodes.

However, Yagi antennas are generally narrowband, often optimized for a single frequency range. To cover multiple bands, several Yagi antennas may be required, which increases deployment complexity. They are also highly directional, meaning alignment and ongoing calibration may be necessary.

3.1.2.1.3. Dipole Antennas

Dipole antennas are a simple, cost-effective baseline option. They are relatively broadband compared to narrowband designs and provide near-omnidirectional coverage, making them a good fit for capturing general RF activity without extensive setup requirements. Their low cost and ease of fabrication also make them appealing for a system intended to be modular and scalable.

On the downside, dipoles typically offer lower gain than directional antennas, which limits detection range and signal clarity. In noisy environments or long-range applications, this can reduce effectiveness. Despite this, dipoles are practical for initial deployment or as supplementary receivers in a multi-antenna diversity configuration.

3.1.2.1.4. Phased Array Antennas

Phased arrays represent an advanced solution, consisting of multiple elements that can be electronically steered. This allows for rapid beam steering and adaptive focusing, providing excellent performance in detecting and tracking RF sources across wide areas. They can also simultaneously process multiple signals, making them highly effective in diversity and multi-node systems.

The tradeoff is complexity and cost. Phased arrays require advanced signal processing, calibration, and additional hardware, which may exceed the budget and simplicity goals of this project. While they offer unmatched precision and adaptability, phased arrays are best suited for high-end or specialized deployments rather than large-scale, low-cost networks.

3.1.2.1.5. Monopole Antennas

Monopole antennas are another low-cost, compact option. Their omnidirectional nature provides 360° reception, making them effective for baseline coverage in diversity setups. They are easy to mount, require little alignment, and work well across a range of frequencies depending on their physical dimensions.

The primary drawback is limited gain and directivity, which can reduce range and performance in weak-signal conditions. Monopoles are often used in combination with other antennas rather than as a standalone solution, adding general coverage at low cost.

3.1.2.1.6. Microstrip Patch Antennas

Microstrip patch antennas are lightweight, compact, and relatively inexpensive, making them well-suited for portable or concealed diversity nodes. They can be designed for specific frequency bands, enabling tailored monitoring of certain signals (e.g., VHF aviation comms or UHF transponders). Their flat profile makes them ideal for integration into enclosures.

However, they typically have narrower bandwidth and lower gain than other antenna options, limiting their effectiveness in wideband applications. This makes them better suited as a specialized supplement to broader-coverage antennas within the system.

3.1.2.1.7. Antenna Comparison

Table 3.3 Antenna Types

Antenna Type	Frequency Range	Gain	Directional	Size	Advantages	Limitations
LPDA	Wide	High	Yes	Larger	Wide bandwidth, high gain	Larger size, complex positioning
Yagi-Uda	Narrow to medium	High	Yes	Moderate	High gain, long-range detection	Requires precise alignment
Dipole	Medium	Med	No	Small	Simple, low cost, omnidirectional	Lower range and gain
Monopole	Medium	Low	No	Small	Low cost, omnidirectional	Lower gain and range
Microstrip Patch	Narrow	Med	Yes	Very small	Compact, easy to integrate	Lower gain, limited range

For our SDR diversity project, we selected dipole antennas as the baseline hardware for all three nodes. The primary driver of this choice was cost and simplicity. Since the project focuses on evaluating the effectiveness of signal diversity techniques rather than maximizing long-range detection, dipoles provide a practical, affordable solution. They are one of the most common antenna designs, consisting of two conductive elements, and can be easily sourced or even fabricated at very low cost. This makes them an excellent fit for the early testing stages of our system, where uniformity, repeatability, and budget-conscious design are more important than absolute performance.

Dipole antennas also offer the benefit of being broadly omnidirectional in the horizontal plane, which is useful for our application. Because our testing environment is close-range and controlled, highly directional antennas such as Yagis or log-periodics would not provide significant additional value. Instead, an omnidirectional antenna ensures consistent signal reception from multiple angles, making it easier to evaluate how the system performs under realistic multipath and interference conditions. This helps us focus on the diversity algorithms and SDR processing chain rather than introducing alignment errors or complex antenna steering requirements.

Another advantage of dipoles is their simplicity of integration with the SDR hardware. They can be directly connected to the SDR inputs with minimal need for specialized matching networks or calibration, reducing complexity in the setup process. For a project where three antennas need to be deployed identically, this ease of deployment ensures consistency across all test nodes, which is critical for meaningful performance comparisons. The compact size of dipoles also makes them highly portable and easy to arrange in different test configurations, whether colocated or spread across a short baseline.

Of course, dipole antennas come with trade-offs. Their gain and range are limited compared to more directional designs, and in noisy environments they may pick up more unwanted signals. This means that while they are sufficient for controlled, short-range experiments, they would not be suitable for large-scale or long-distance deployment in later project phases. However, for the purpose of validating our diversity system, these limitations are actually beneficial, as they allow us to stress-test the DSP algorithms under realistic conditions of interference and multipath without artificially boosting the signal quality.

In summary, the dipole antenna strikes the right balance of cost, ease of use, and suitability for controlled testing. Using three identical dipoles ensures that our diversity experiments remain fair and consistent, while their broad coverage and simple design provide a robust foundation for evaluating SDR-based signal selection and combining methods. Once the algorithms are validated with dipoles, the system can later be scaled to incorporate higher-performance antennas such as LPDAs or Yagis for longer-range, real-world applications.

3.1.2.2 Software Defined Radios

For our Radio Diversity System, the SDRs form the backbone of signal acquisition and processing. Unlike traditional spectrum analyzers, SDRs digitize incoming RF signals and allow for real-time processing through software-defined algorithms. This flexibility is essential for

implementing diversity techniques, where multiple signals from different antennas are compared to determine the strongest or cleanest reception path.

We selected SDR hardware because it offers a balance of affordability, performance, and modularity. Low-cost models such as the RTL-SDR Blog V4 can cover frequencies from 500 kHz to 1.7 GHz, which includes the VHF airband and ADS-B at 1090 MHz. For higher performance, mid-tier SDRs like the Airspy R2 or SDRplay RSP1A can extend dynamic range and improve signal-to-noise ratios. Multiple SDR units are used simultaneously, each connected to a separate antenna, allowing our system to capture diverse perspectives of the same signal environment.

One of the key advantages of SDRs is their ability to operate across wide frequency ranges while maintaining a compact and power-efficient form factor. They are portable and easily integrated into our Raspberry Pi-based processing unit. This ensures our diversity system can be deployed in remote environments where size, weight, and power consumption are critical considerations. SDRs also benefit from strong open-source software support, including tools such as GNU Radio, SDR#, and custom Python-based DSP scripts, enabling us to implement advanced diversity algorithms without requiring expensive proprietary software.

However, SDRs also come with some limitations. Entry-level models may have limited instantaneous bandwidth and dynamic range compared to laboratory-grade analyzers. This means additional front-end components, such as low-noise amplifiers (LNAs) and bandpass filters, may be required to optimize performance. Despite these challenges, the low cost, modularity, and software-driven flexibility of SDRs make them an excellent fit for our diversity-based system.

3.1.2.3 Setup with SDRs and Antennas

Integrating the antennas with the SDRs requires careful attention to signal integrity, component selection, and best practices in RF system design. A typical configuration consists of:

Antennas → Low-Loss Coaxial Cable → Optional LNAs/Filters → SDR Inputs → Raspberry Pi (DSP) → Output to Laptop/Monitoring Interface

For our application, each SDR is connected to a dedicated antenna positioned with spatial separation to maximize diversity benefits. This separation ensures that each antenna experiences different fading conditions, allowing the system to intelligently select the best-quality signal. Coaxial cable selection is critical, as high-frequency losses can significantly degrade performance; cables such as LMR-400 or equivalent low-loss options are recommended for minimizing attenuation.

To enhance signal quality, optional front-end components are incorporated:

- **Low-Noise Amplifiers:** Boost weak aircraft signals before they are attenuated in the cable. Placing LNAs near the antenna maximizes sensitivity.
- **Bandpass Filters:** Narrow the frequency range to reduce out-of-band interference and improve signal-to-noise ratio. For example, a 1090 MHz filter can be used for ADS-B

monitoring.

- Attenuators: Protect the SDR from overload in areas with very strong signals.

The Raspberry Pi acts as the central processor, running DSP algorithms that synchronize the SDR inputs and apply selection combining or other diversity methods. Software settings such as sample rate, bandwidth, and gain control are tuned depending on the frequency band of interest. A monitoring laptop can access the Raspberry Pi either directly or remotely to visualize processed data.

Environmental considerations are also important. Antennas should be mounted with clear line-of-sight and minimal obstructions. Weatherproof housings and proper grounding protect the system during long-term deployments in remote or harsh conditions. This setup ensures reliable data collection and robust operation in the field.

3.1.2.4 Antenna Selection

For this project, dipole antennas were selected as the baseline for all three diversity channels. The decision was based on their low cost, ease of deployment, and suitability for controlled, short-range experiments. Using three identical dipoles ensures consistency across tests and provides a fair basis for evaluating the SDR's diversity performance. Their omnidirectional pattern allows reception of signals from multiple angles, which is ideal for assessing the effectiveness of selection and combining techniques in multipath environments.

Each dipole connects directly to an SDR input through a coaxial cable, forming a straightforward antenna-to-receiver chain. To improve signal quality, optional low-noise amplifiers (LNAs) may be added before the SDR inputs to boost weak signals, while bandpass filters can be used to suppress out-of-band interference when focusing on specific frequency ranges. In cases where very strong signals are present, simple attenuators can also be included to prevent SDR input overload. These additions are not mandatory for initial testing but provide flexibility for refining performance as experiments scale.

By combining the dipole's simplicity with the SDR's digital signal processing capabilities, the system achieves a clean, modular design that is both affordable and flexible. This setup allows us to focus on testing and validating diversity techniques—such as strongest-link selection or combining methods—without unnecessary complexity in the antenna hardware. As the project matures, the system can later expand to higher-gain or directional antennas, but for initial proof-of-concept, dipoles remain the most practical and scalable choice.

3.1.3 Software Defined Radio

3.1.3.1 Choosing an SDR

The choice of SDR hardware is critical to the success of the radio diversity project, as it determines how reliably and accurately the system can capture RF signals from the antennas. Because the focus of this project is testing diversity techniques with three antennas, the system requires either multiple independent SDR units or a single multi-input SDR that can handle

simultaneous channels. The tradeoffs between cost, complexity, and technical performance play a large role in this decision, especially since the project pivot and funding reduction have forced the team to prioritize affordability and ease of implementation over professional-grade performance.

3.1.3.1.1 Multiple Single Input SDRs

One option is to deploy three single-input SDRs, such as the widely available RTL-SDR v3. Each SDR would connect to a separate antenna, allowing the Raspberry Pi to capture and process three independent RF streams. The RTL-SDR v3 is particularly attractive because of its low cost (around \$30 per unit), broad frequency coverage up to 1.7 GHz, and strong community support. This makes it an ideal choice for proof-of-concept diversity experiments, where absolute synchronization is less important than demonstrating the principles of selection combining, switch combining, or simple diversity gains.

The benefits of this approach are clear: it is cost-effective, modular, and simple to scale. If one unit fails, it can be replaced without impacting the others, and additional units can be added easily if the system needs to expand. However, there are drawbacks. Each SDR has its own local oscillator, which means that their clocks are not synchronized. This makes precise techniques like time-difference-of-arrival (TDOA) localization more difficult. USB bandwidth management is another concern, since the Raspberry Pi will need to process data from three devices simultaneously. Despite these limitations, multiple single-input SDRs offer the most practical balance of affordability and functionality for the current project stage.

3.1.3.1.2 Multi Input SDR

In evaluating multi-input SDRs for our radio diversity project, it became clear that the technical benefits of synchronized channels are valuable, but cost and availability are key deciding factors given the recent funding cuts. Multi-input SDRs offer two or more receiver channels in a single device, typically with shared clocking or internal synchronization. This simplifies or eliminates problems of drift, phase misalignment, and timing differences between separate SDR units, which are especially important if we ever move toward more advanced techniques like beamforming, coherent combining, or time-difference-of-arrival (TDOA).

Real products in this space include the LimeSDR Mini 2.0, which offers dual-channel input with wide bandwidth (around 40 MHz), full-duplex operation, and frequency coverage up to several GHz. These features make it technically very strong. However, its price (~US \$300-\$400 or more depending on source and availability) often puts it close to or above the cost of purchasing three lower-end SDRs. There are occasionally listings on Amazon or other marketplaces for dual-input SDRs that approach competitive pricing, but they often come with risks: lower manufacturing quality, less reliable support / firmware, more limited documentation, and sometimes inflated “dual input” claims that are not well synchronized.

While searching Amazon, it is possible to find SDR devices advertised with “dual antenna input” or “multi-band input,” which might look like cheap multi-input SDRs. Some examples like the Malachite DSP2 SDR on Amazon have dual antenna inputs for reception over a wide frequency span. However, the labeling often refers to dual antenna ports, which does not necessarily mean fully independent and synchronized RF chains. For diversity, what matters is whether the two inputs are phase coherent or how much drift etc. plays into the system. Regular RTL-SDR dongle bundles like the Nooelec or NESDR SMARt series also appear frequently. These are low-cost, single input/dongle devices. Having three of these is cheaper, and easier to mass-replace or scale, than one dual-input SDR in some cases.

These cheaper Amazon offerings may capture the “two antennas into one box” concept, but often trade off coherence, shared clocking, or driver stability. That means that for diversity testing, some Amazon dual-port SDRs might be acceptable if performance is understood and limitations accounted for.

3.1.3.2 Connection to Raspberry Pi

Regardless of whether multiple single-input SDRs or a single multi-input SDR is used, integration with the Raspberry Pi follows the same general model. The antennas connect to the SDR via coaxial inputs, and the SDR digitizes the incoming RF signals. These digitized streams are then sent over USB to the Raspberry Pi, where digital signal processing (DSP) routines and diversity algorithms are implemented. With low-cost devices such as the RTL-SDR v3, the USB 2.0 interface is sufficient to support the limited sampling rate and bandwidth. In contrast, higher-end SDRs often require USB 3.0 or PCIe connections, which exceed the capabilities of the Raspberry Pi. For this reason, low-cost SDRs are far easier to integrate into the current project architecture, where keeping the system lightweight and portable is a primary goal.

Optional components such as low-noise amplifiers (LNAs) or band-pass filters could be inserted between the antennas and SDRs to improve signal quality, but these are not strictly necessary for short-range testing. For now, the design emphasizes simplicity, with a direct path from antennas to SDRs to the Raspberry Pi. This keeps costs low while still allowing the diversity experiments to be carried out effectively.

3.1.3.3 Final Selection

Initially, the project’s predecessor system used the Signal Hound BB60D spectrum analyzer, a professional-grade device costing over \$3,000. This instrument offered outstanding bandwidth and precision, making it ideal for long-range passive aircraft detection. However, with the shift in project scope toward radio diversity testing and the simultaneous reduction in funding, the use of such high-end equipment is no longer feasible. Instead, the emphasis has turned toward low-cost, modular SDRs that can still enable meaningful diversity experiments without straining the budget.

For this reason, the team has selected three RTL-SDR v3 units as the final hardware choice. While these devices lack the synchronization and high-resolution capabilities of more advanced SDRs, they are more than sufficient for demonstrating diversity gains in a short-range testbed. Their low cost makes them accessible within the new funding limits, and their widespread documentation ensures that integration challenges can be overcome quickly. The RTL-SDR setup preserves flexibility, allowing the system to expand or upgrade to multi-input SDRs in the future if additional funding becomes available. This decision reflects the project's pivot toward cost-conscious experimental validation, while still providing a pathway for future scalability and technical growth.

In practice, the three SDRs will be connected to the Raspberry Pi through a powered USB hub. This configuration is necessary because the Pi's onboard USB ports alone cannot reliably deliver the current required by multiple SDRs operating simultaneously. A powered hub not only ensures stable voltage and current to each dongle, but also helps distribute bandwidth across multiple USB lanes, reducing the chance of dropped samples or data bottlenecks during high-rate streaming. Careful hub selection is therefore part of the system design, with attention given to total current output, shielding, and hub chipset compatibility with the Pi. This modular USB-based architecture keeps the system inexpensive and easily expandable, while also highlighting the trade-off between affordability and performance in the project's current stage.

3.1.4 Raspberry Pi Touchscreen

3.1.4.1 Choosing a Touchscreen

Selecting a touchscreen for this project involves balancing cost, responsiveness, compatibility, and physical size. Key factors include native support for the Raspberry Pi, low-latency display updates, and adequate resolution for displaying signal plots and node metrics. Several options exist, including official Raspberry Pi touchscreens, third-party DSI-compatible displays, and HDMI-connected monitors with touch overlays. The official Raspberry Pi 7-inch touchscreen offers direct DSI connection, which reduces wiring complexity and provides consistent power from the Pi itself. It also supports multi-touch gestures, enabling more sophisticated GUI interactions. Third-party DSI screens may be cheaper, but often have lower documentation support, inconsistent driver compatibility, or reduced touch accuracy, increasing integration effort. HDMI touchscreens offer higher resolution options, but require additional USB connections for touch functionality, external power sources, and more enclosure space, which complicates portability and system simplicity.

Other considerations include durability and environmental tolerance. Since the touchscreen may be handled frequently during testing, a robust surface resistant to scratches or smudges is important. Display brightness and contrast must also be sufficient for lab lighting conditions, as well as potential outdoor testing where glare could be an issue. Finally, cost is a major factor in

the current budget-conscious environment; the selected touchscreen must provide sufficient functionality for experimentation without significantly increasing project expenses.

3.1.4.2 Integration with Raspberry Pi

The chosen touchscreen connects to the Raspberry Pi via the DSI port, allowing both display data and touch input to be transmitted over a single interface. This direct connection minimizes latency and ensures responsive touch interactions, which is crucial when adjusting SDR settings or observing live signal metrics in real time. A custom GUI is implemented using Python libraries such as Tkinter or PyQt5, providing controls for SDR broadcasts, antenna switching, and signal visualization. Operators can tap to select active SDRs, swipe through graphical plots, or adjust broadcast parameters without requiring a separate monitor or keyboard, streamlining the testing process.

Integration also considers software and system performance. The touchscreen requires proper calibration and driver installation to ensure accurate touch mapping and consistent display output. Additionally, the Raspberry Pi must handle GUI rendering alongside SDR data processing without introducing noticeable lag. Efficient programming practices, such as updating plots at limited intervals and optimizing background tasks, help maintain smooth operation. This integration approach enables operators to perform iterative testing, monitor signal quality, and diagnose system issues on a single compact device.

3.1.4.3 GUI Design and Layout

The touchscreen GUI is a central feature for monitoring and controlling the radio diversity testbed, providing operators with clear visual feedback and interactive controls. The interface is designed to display real-time signal metrics from the three SDR nodes, highlight the strongest signal, and allow switching between different diversity modes. At the top of the screen, a status bar provides global information, including system uptime, active diversity mode, and overall signal quality indicators.

The main display area is divided into three signal panels, each representing one of the SDR nodes. Each panel shows real-time plots of signal strength (RSSI), signal-to-noise ratio (SNR), and time-of-arrival (TOA) data. A dynamic highlight or color-coding indicates which node currently has the strongest signal, helping operators quickly identify the optimal reception path without manually inspecting raw data. These visual cues are particularly valuable when testing spatial diversity configurations, where node performance can vary depending on antenna placement or environmental conditions.

Below the signal panels, mode selection buttons allow users to switch between Frequency Diversity, Time Diversity, Spatial Diversity, and Manual Mode. In Frequency, Time, or Spatial Diversity modes, the system automatically selects the best signal based on the respective

algorithm, updating the visual highlights in real time. Manual Mode enables the operator to tap on any of the three signal panels to force selection of a specific SDR node, providing flexibility for testing edge cases or conducting controlled experiments. Each mode button is visually distinct, with active mode clearly highlighted to reduce confusion during operation.

The GUI also includes a small control panel for additional system adjustments. Operators can set frequency ranges, adjust broadcast parameters for testing, and view diagnostic messages from the Raspberry Pi or SDR nodes. Touch interactions are designed for responsiveness, allowing real-time updates without lag. Optional features, such as logging or exporting data, can be toggled from a settings menu accessible via the corner of the screen, keeping the main interface uncluttered while preserving advanced functionality for detailed experiments.

Overall, this GUI design balances usability, visibility, and interactivity. It allows operators to monitor multiple SDR signals simultaneously, quickly identify the strongest signal, switch diversity modes, and intervene manually when needed. By providing an intuitive and visually clear interface, the touchscreen becomes a powerful tool for experimentation, enabling rapid testing of broadcast strategies, node configurations, and radio diversity performance.

3.1.4.4 Hardware Options

Selecting the appropriate touchscreen display is crucial for the functionality and usability of the radio diversity testbed. Given the project's emphasis on real-time signal monitoring and mode selection, the touchscreen must offer sufficient resolution, responsive touch capabilities, and compatibility with the Raspberry Pi. Below are several options that meet these criteria.

Table 3.4 Touchscreen Options

Display Model	Resolution	Touch Type	Interface	Power Source	Price Range
Official Raspberry Pi 7" Touchscreen Display	800×480	Capacitive	DSI	Raspberry Pi	\$60.00
Waveshare 7-inch Capacitive Touch Screen LCD	1024×600	Capacitive	HDMI	USB	\$43.99
Elecrow 7-inch DSI Touchscreen Display	800×480	Capacitive	DSI	Raspberry Pi	\$56.48

Elecrow 7-inch Capacitive Touchscreen Monitor	1024×600	Capacitive	HDMI	USB	\$55.11
Freenove 7-inch Touch Display for Raspberry Pi	1024×600	Capacitive	DSI	Raspberry Pi	\$49.99

3.1.4.5 Final Selection

To support local control, visualization, and diagnostic capabilities for our radio diversity-based aircraft detection system, we selected a 3.5-inch Raspberry Pi-compatible touchscreen display as the Human-Machine Interface (HMI). This compact display offers a resolution of 480×320 pixels, which is sufficient for presenting system data such as RSSI values, hop status, node synchronization, and detection logs in a clear, structured interface. It connects via GPIO and SPI, ensuring minimal port usage and leaving USB and HDMI ports available for critical peripherals such as LoRa transceivers, GPS modules, and external data storage. This streamlined integration is essential for maintaining compact, modular hardware design in remote deployments.

The resistive or capacitive touchscreen interface, depending on the model selected, enables simple tap navigation for users in the field—supporting quick access to system health, detection history, or reboot controls without the need for a mouse or keyboard. The low power consumption of approximately 0.1–0.5W makes it highly compatible with our solar-battery power system, enabling reliable operation without draining the energy budget in off-grid installations. The screen’s slim profile makes it ideal for panel mounting or portable diagnostic kits, and its small footprint allows integration into compact sensor housings without requiring enclosure redesign.

We plan to implement a Python-based GUI using PyQt5 or Tkinter, optimized for smaller screen sizes and touch interaction. The interface will display key metrics such as packet signal strength, hop integrity, and diversity scoring, along with timestamped detection logs and basic manual override options. This setup is well-suited to run on the Raspberry Pi 3 Model B, which provides sufficient computing resources (1.2 GHz quad-core processor and 1GB RAM) to support both the GUI and background detection services concurrently.

Its DSI interface simplifies wiring and power requirements, ensuring a compact and portable setup suitable for short-range lab experiments. The touchscreen’s resolution and multi-touch capability are sufficient for displaying SDR node statuses, signal metrics, and control options, while its official driver support guarantees reliable integration with the Pi. Third-party

alternatives were considered but eliminated due to potential driver issues, inconsistent touch accuracy, and additional integration complexity.

This touchscreen aligns with the project’s focus on cost-effective, modular testing equipment. It enables rapid experimentation with SDR broadcasts and antenna placements, provides a user-friendly interface for operators, and maintains system portability. While higher-end displays could offer larger screens or higher resolution, the official 7-inch option strikes the optimal balance between performance, usability, and affordability, supporting both current lab-scale experiments and future potential expansions without exceeding the reduced budget.

However, if budget constraints are a significant consideration, the Waveshare 7-inch Capacitive Touch Screen LCD provides a higher resolution and is priced competitively. Its HDMI interface offers flexibility in connectivity, though it may require additional considerations for power supply.

3.2. Power Supply System

With the primary processing components and radio diversity-based detection architecture now selected, we can move forward with designing a power delivery system capable of supporting sustained, autonomous operation in remote environments. Our system consists of multiple spatially distributed sensor nodes, each equipped with an SBC (such as a Raspberry Pi), an RF front-end for passive signal capture, and wireless communication modules for internal data sharing and uplink via Starlink SATCOM. Since these units will often be deployed in locations without access to traditional power infrastructure, a reliable and energy-efficient off-grid power solution is essential.

Initial power profiling estimates the peak consumption of each detection unit at approximately 45W, accounting for the SBC, RF signal processing modules, and wireless transceivers. However, as with most embedded systems, it is unlikely that all components will operate at full load simultaneously. Power usage will vary throughout the detection cycle—spiking during transmission and processing but remaining low during idle or sleep phases. To strike the right balance between reliability, scalability, and cost, we have chosen a solar panel and battery combination to serve as the system’s primary power source.

Specifically, each node will use a 50W solar panel to collect energy during daylight hours, charging a deep-cycle lithium or LiFePO₄ battery that provides continuous power during the night or low-light conditions. This configuration supports 24/7 autonomous operation with minimal maintenance, making it ideal for long-term field deployments. Additionally, this solution is compact and modular, allowing the entire system—solar panel, charge controller, and battery—to be mounted on lightweight, weather-resistant enclosures alongside the RF hardware.

The system also features multi-voltage power distribution to accommodate the diverse requirements of each component:

- 3.3V for low-power MCUs or wireless modules (e.g., LoRa, BLE)

- 5V for the Raspberry Pi SBC, USB peripherals, and signal conditioning circuits
- 12V for higher-power devices such as long-range radios or satellite transceivers

A DC-DC buck/boost converter module will be used to regulate voltage levels with high efficiency, minimizing power loss across the system. These converters are selected based on their ability to handle temperature extremes and fluctuating solar input—both common in remote deployments.

To support planning and integration, Table 3.4 outlines the estimated peak power consumption per component, helping us size the battery capacity and solar panel rating more accurately. This ensures that each detection unit in our radio diversity network remains powered continuously, even under intermittent weather conditions.

3.2.1 Batteries

3.2.1.1 Choosing a Battery

In remote regions where conventional radar is not viable, our node system relies on solar-powered units to operate autonomously for extended periods. A key component of this system is an efficient, long-lasting rechargeable battery that can store enough energy to power RF receivers, processing units, and satellite communication equipment. Since these units will be deployed in isolated areas with varying weather conditions, the selected battery must be high-capacity, durable, and optimized for solar charging.

For our applications, the primary criteria are reliability, efficiency, and long-term performance, as our nodes will be out in the wilderness, and should not require constant maintenance. Since the system will likely be powered by solar energy, the battery must be capable of efficiently storing and discharging energy in varying sunlight conditions, ensuring continuous operation even during periods of low light or at night. The battery must also have a high cycle life to withstand frequent charging and discharging without significant degradation, as the system may need to operate for extended durations with minimal maintenance. Additionally, the battery should be able to handle temperature fluctuations, as the detection unit could be deployed in diverse environmental conditions. We can also consider size and weight as semi-important factors for remote deployments, where space may be limited and lightweight components are desirable. Finally, cost-effectiveness is a key consideration, given the goal of maintaining a low overall cost for the detection unit while ensuring the battery is durable and reliable over time.

All these factors will guide the selection of the most suitable battery technology for this application.

3.2.1.2 Available Battery Options

When selecting a rechargeable battery for our detection nodes, several types of batteries can be considered: Lithium Iron Phosphate (LiFePO₄), Lithium-Ion (Li-Ion), Sealed Lead Acid (SLA), and Gel batteries. LiFePO₄ is known for its long cycle life, high safety, and stability, making it an excellent choice for long-term, reliable energy storage in outdoor applications. Li-Ion

batteries offer higher energy density and lighter weight compared to other options, which is advantageous for applications where space and weight are constraints. SLA batteries, while heavier and less efficient, are cost-effective and widely available, making them a budget-friendly option, though they tend to have shorter lifespans. Gel batteries are similar to SLA but feature a gel electrolyte that improves safety and performance in extreme conditions, though they are typically more expensive and heavier. Each of these battery types offers trade-offs in terms of cost, performance, and lifespan.

3.2.1.3 Battery Option Chart

Table 3.5 Battery Types

Battery Type	Capacity (Ah)	Voltage (V)	Cycle Life	Efficiency	Pros	Cons
Lithium Iron Phosphate (LiFePO4)	100	12	3000-5000	95%	High cycle life, lightweight, deep discharge	Higher cost upfront
Lithium-Ion (Li-Ion)	100	12	2000-4000	90-95%	Compact, good efficiency	Slightly lower lifespan than LiFePO4
Sealed Lead Acid (SLA)	100	12	500-1000	80-85%	Lower cost, easy to find	Heavy, shorter lifespan
Gel Battery	100	12	600-1200	85-90%	Maintenance-free, deep cycle	More expensive than SLA, lower lifespan than LiFePO4

This chart compares four types of rechargeable batteries—Lithium Iron Phosphate, Lithium-Ion, Sealed Lead Acid, and Gel Batteries—across key parameters such as capacity, voltage, cycle life, efficiency, and pros and cons. All options in the chart have similar capacities (100 Ah) and voltage (12 V), but they differ significantly in cycle life, efficiency, and cost-effectiveness. LiFePO4 offers the highest cycle life (3000-5000 cycles) and efficiency (95%), making it ideal for long-term, reliable use, though it comes with a higher upfront cost. Li-Ion provides good efficiency and compactness, with a slightly lower cycle life than LiFePO4. SLA and Gel batteries are much more affordable but come with significantly shorter cycle lives and lower efficiency. While Gel batteries offer maintenance-free operation, their lifespan is still shorter compared to LiFePO4.

For the detection nodes, a battery with a high cycle life and reliability in varying environmental conditions is crucial. Given that LiFePO₄ offers the best combination of long cycle life, efficiency, and deep discharge capabilities, it is likely the most suitable choice despite the higher initial cost. The long-term performance and reliability it provides outweigh the initial investment, especially for a system that needs to operate continuously with minimal maintenance.

3.2.1.4 LiFePO₄ Battery Choices

The following table provides a comparison of several 12V 100Ah Lithium Iron Phosphate (LiFePO₄) batteries, outlining their prices, key features, and where we can find them. These options are suitable for applications requiring reliable energy storage, such as in a solar-powered passive aircraft detection unit. Each battery varies in terms of cost, lifespan, and added features, allowing for different trade-offs based on specific needs and budget constraints.

Table 3.6 Battery Models

12V 100Ah LiFePO ₄ Lithium Battery		
Brand	Price (USD)	Key Features
Redodo	\$167.88	Budget-friendly, suitable for solar
LiTime	\$199.99	Deep cycle, designed for energy storage applications.
Eco-Worthy	\$209.99	Compact design, lifespan up to 15,000 cycles.
Battle Born	\$749.99	Known for durability and versatility, tailored for RVs and off-grid systems.
Renogy	\$709.99	Smart features with Bluetooth connectivity for monitoring.
EcoFlow	\$250.00	Low lifetime cost, 6,000 cycle lifespan.
WattCycle	\$159.99	A+ grade cells, up to 20,000 cycles.
ExpertPower	\$329.99	Flat discharge curve, lifespan of 2,500 to 7,000 cycles.

Considering the balance of price, lifespan, and reliability, the LiTime 12V 100Ah LiFePO₄ Lithium Battery emerges as the best option. Priced at \$199.99, it offers a solid combination of affordability, a reasonable lifespan, and reliability for long-term use. While more expensive

options like Battle Born and Renogy offer additional features, the LiTime battery provides a good balance without breaking the budget, making it a great fit for our system..

3.2.2 Solar Panels

3.2.2.1 General Types of Panels

When selecting a solar panel for our detection system, several factors need to be considered, including the local environmental conditions, the required power output, panel efficiency, lifespan, cost, and physical space for installation. Since the system is intended for remote areas with limited infrastructure, the ideal solar panel should be highly reliable, efficient in diverse weather conditions, and affordable to maintain. Given that the system will be powered continuously, energy efficiency will be critical to ensure long-lasting performance without requiring frequent maintenance. Additionally, ease of installation and durability are important, as the system will be deployed in harsh and often unpredictable environments, such as remote border areas.

There are four types of solar panels we will consider: Monocrystalline, Polycrystalline, Thin Film, and PERC. In selecting the type of panel, we must also account for the geographic location and the amount of available sunlight. For instance, if the system is deployed in areas with consistent sunlight, high-efficiency panels might be prioritized. Conversely, if the system is located in regions with partial shading or low sunlight, we may prioritize panels that perform better under such conditions, even if they come at a slightly higher cost. Similarly, since these systems may operate in large networks, scalability and the ability to quickly deploy additional panels will also influence the type of solar panels selected.

3.2.2.2 Pros and Cons

For our Power Supply System, we can consider several types of solar panels, each with unique advantages and trade-offs. Understanding the underlying technology and performance characteristics of each option is essential for selecting the best fit for our remote deployment.

Monocrystalline panels are known for their high efficiency, with typical energy conversion rates up to 22%. These panels are made from single-crystal silicon, which is grown in a way that allows electrons to flow more freely, enhancing performance. Their compact, space-efficient design makes them ideal for installations in areas where space is limited, such as remote monitoring stations. Monocrystalline panels also have a long lifespan, often exceeding 25 years, and perform excellently in low light conditions, ensuring continuous energy generation even during overcast or shaded periods. However, the main drawback is their higher cost compared to other panel types. Additionally, their fragility during installation can present challenges, as the panels can be brittle and require careful handling.

Polycrystalline panels are made from silicon that has been melted and poured into molds, forming multiple crystals. This process is less expensive and results in a lower cost compared to monocrystalline panels. They typically have an efficiency rate between 15% and 17%, which is sufficient for general applications, though their lower efficiency means that a larger surface area is required to generate the same amount of power as monocrystalline panels. These panels are still a solid choice for systems where cost is a significant concern, but their larger size and

relatively lower efficiency make them less ideal for applications requiring space-saving solutions or higher power output. Polycrystalline panels also have a lifespan of around 20–25 years, but their performance under low-light conditions is not as strong as that of monocrystalline panels.

Thin-film panels, made from layers of semiconductor materials such as cadmium telluride or amorphous silicon, are lightweight and flexible. This makes them versatile, allowing integration into unconventional surfaces like roofs or walls. They are typically lower in cost, which may seem appealing for budget-constrained projects. However, thin-film panels come with significant downsides. Their efficiency is the lowest of the options we considered, typically around 10% or less, meaning that much larger surface areas are required to meet energy demands. Furthermore, their shorter lifespan (approximately 10–15 years) and poor performance in low-light conditions make them less suitable for our use case, where reliability and efficiency are key to maintaining system operation in remote locations over an extended period.

PERC (Passivated Emitter and Rear Contact) panels are a more advanced version of monocrystalline panels. They employ an additional layer on the back of the cell to reduce energy losses and improve overall efficiency, reaching up to 22%. PERC technology enhances performance, particularly in low light and high-temperature conditions, making these panels more efficient and reliable than conventional monocrystalline panels. However, the increased efficiency comes with a higher cost, and the panels require more specialized installation and handling due to the advanced technology involved. This makes them a good option for situations where efficiency is paramount, but the added complexity and cost may not justify the benefits in every case.

Table 3.7 Solar Panel Types

Type of Panel	Efficiency	Cost Range	Lifespan	Best Use Case
Monocrystalline	18%-22%	High (\$0.80-\$1.00/W)	25+ years	High efficiency, small footprint, remote areas with high sunlight
Polycrystalline	15%-17%	Medium (\$0.60-\$0.80/W)	20-25 years	Budget-conscious solutions, larger area available
Thin Film	10%-13%	Low (\$0.50-\$0.70/W)	10-15 years	Large area installations, less critical for space
PERC	20%-22%	Medium-High (\$0.90-\$1.10/W)	25+ years	High-efficiency needs in low light conditions

For our Power Supply System, we have already decided that our best battery option is a 12V 100Ah LiFePO4 battery. With that in mind, along with the pros and cons of each kind of solar panels, monocrystalline solar panels are the best choice for us. Monocrystalline panels offer the highest efficiency, typically between 18% and 22%, which means they generate more power per square meter compared to other types like polycrystalline and thin-film panels, while being lower cost than the comparable PERC panels. This is particularly beneficial for our system, as maximizing energy production in a limited space is crucial, especially in remote and difficult-to-access locations as well as to avoid detection from the aircrafts. Monocrystalline panels are also known for improved performance in low-light conditions, making them a great option for areas with inconsistent sunlight or partial cloud cover. In such environments, these panels will continue to provide reliable power, ensuring the battery is charged efficiently.

Additionally, these panels boast a long lifespan of over 25 years, offering durability and reliability for a system that will likely be deployed in harsh, hard-to-reach environments where maintenance is limited. Furthermore, monocrystalline panels output a voltage that typically falls between 18V–22V, making them an excellent fit for efficient charging of our 12V battery system, when paired with a charge controller. Though polycrystalline are a cheaper option, the efficiency is lower (15-17%) making them less than ideal for low-light locations, meaning we would have to purchase more panels, which will cost a similar price to the more expensive monocrystalline panels. Thin-film panels are not viable due to their poor efficiency and shorter lifespan. PERC panels, though superior in efficiency, involve higher costs and specialized handling, which is not necessary for our application. Ultimately, monocrystalline panels strike the best balance between performance, cost, and long-term reliability, ensuring that our system can operate autonomously in remote regions with minimal maintenance. Thus, we propose using a monocrystalline panel for the design

3.2.2.3 Specific Models

Table 3.8 Solar Models with Features

Brand	Model	Cost	Key Features
Renogy	RNG-100D	\$120	100W, Monocrystalline, Compact, High-Efficiency (21%)
Goal Zero	Nomad 100	\$300	100W, Monocrystalline, Foldable, Portable, Waterproof
SunPower	SPR-X22-370	\$350	370W, Monocrystalline, PERC, Industry-leading efficiency (22%)
Eco-Worthy	Solar Panel 100W	\$100	100W, Polycrystalline, Durable, Cost-effective

ALLPOWERS	AP-100W	\$160	100W, Monocrystalline, Lightweight, Waterproof, Foldable
------------------	---------	-------	--

3.2.3 Charge Controllers

3.2.3.1 Purpose of Charge Controller

A charge controller is a vital component of the solar power system. It regulates the voltage and current coming from the solar panels to ensure the battery is charged efficiently and safely. Charge controllers prevent overcharging, deep discharging, and allow the battery to last longer by controlling the charge flow. When evaluating charge controller options, several key factors need to be considered, including system size, efficiency, compatibility with the LiFePO4 battery, and scalability for future expansions.

3.2.3.2 Types of Charge Controllers

There are two primary types:

- PWM (Pulse Width Modulation) Charge Controllers: More affordable, suitable for smaller systems with lower power needs. However, less efficient than MPPT controllers.
- MPPT (Maximum Power Point Tracking) Charge Controllers: More expensive but highly efficient, especially in low-light conditions. MPPT controllers are ideal for larger systems where performance and energy efficiency are paramount.

Given our system's requirements (high efficiency, scalability, 12V 100Ah LiFePO4 battery) an MPPT charge controller is the best choice. An MPPT controller ensures that excess voltage from the panels is converted to the optimal charging current for the LiFePO4 battery, maximizing energy efficiency and reducing charging time, even in variable lighting conditions. This makes it particularly valuable for the remote environments where the nodes will have to be deployed. Additionally, an MPPT controller is better suited for future scalability, especially if we expand the system to include more panels. While the higher cost of MPPT controllers is a consideration, the increased efficiency and long-term reliability they offer make them the best investment for our application, ensuring the system remains operational in the most energy-efficient manner.

3.2.3.3 Charge Controller Options

Table 3.9 Charge Controller Models

Brand	Model	Type	Cost	Features	Applications
Renogy	Rover 40A	MPPT	\$125	40A, Efficient in variable sunlight, LCD display	High-efficiency, solar systems with larger panels
Victron	SmartSolar MPPT 75/15	MPPT	\$85	15A, Bluetooth monitoring, High performance	Smaller systems or remote monitoring

AllPowers	10A PWM	PWM	\$30	Affordable, Suitable for 12V systems	Budget systems or small setups
EPEVER	Tracer AN 30A	MPPT	\$75	30A, Waterproof, Battery type flexibility	Remote systems with high solar exposure

3.2.4 Wiring Setup

3.2.4.1 For a Single Unit

For a single unit setup, the solar panel is connected to the charge controller by linking the positive and negative terminals of the solar panel to the corresponding solar input on the charge controller. From there, the charge controller connects to the LiFePO₄ battery. The positive and negative terminals of the charge controller's battery output are connected to the battery, allowing it to charge. Once the battery is fully charged, the detection unit, which serves as the system load, is then connected to the battery to draw power and operate continuously.

3.2.4.2 For Multiple Unit

For multiple units, solar panels can be connected in either a series or parallel configuration, depending on the voltage requirements of the charge controller and the battery. Connecting panels in series increases the voltage, while connecting them in parallel increases the current output, providing flexibility in how the system is scaled. If we plan to expand to a large system, it may be necessary to use multiple charge controllers. Each group of solar panels could be connected to its own controller, ensuring efficient charging and distribution of power. To streamline coordination and ensure the entire system operates seamlessly, a master/slave setup for the charge controllers can be implemented. This approach allows for better monitoring and synchronization across the system, which is especially important in larger installations.

3.2.4.3 Additional Recommendations

Additionally, some important recommendations should be followed to maintain system safety and performance. Overcurrent protection devices such as fuses or circuit breakers should be installed in both the panel-to-controller and controller-to-battery connections to protect against potential short circuits or overcurrent situations. Proper sizing of the solar panels and battery is crucial to meet the system's energy requirements; using a system size calculator can help ensure the panel wattage and battery capacity are well-matched. Wire gauge selection is also vital to minimize voltage loss and safely handle the current. For instance, a 10 AWG wire is typically suitable for connecting the solar panels to the charge controller, while a 12 AWG wire is appropriate for connecting the battery to the system load. By ensuring the correct sizing of components and incorporating safety measures, the system will be more reliable and efficient in both single and multi-unit configurations.

3.2.5 Power Supply Part Selection

In the original system concept, we designed a robust off-grid power supply using Renogy solar components, including the RNG-100D Solar Panel, the Renogy Rover 40A MPPT Charge

Controller, and the LiTime 100Ah 12V LiFePO4 Battery. This setup was intended to ensure long-term autonomous operation of remote detection nodes, even in challenging environments, by combining renewable energy generation, intelligent charge management, and reliable energy storage. The solar-based system provided a sustainable power solution with minimal maintenance needs and strong compatibility between components.

However, due to project funding cuts and a change in scope, the current design no longer requires full off-grid autonomy. Instead, the system is powered using a standard AC-DC supply, which provides consistent energy without the cost or complexity of a dedicated solar system. This approach significantly reduces setup and deployment requirements, while still ensuring reliable operation for testing and proof-of-concept demonstrations.

For components like the Raspberry Pi, which requires a regulated 5V input, a simple step-down converter (buck regulator) can be used if the AC-DC supply provides a higher voltage (e.g., 12V). This ensures safe and stable power delivery to the Pi and other sensitive electronics. Since our testing is conducted in controlled environments with grid access, this streamlined design is both practical and cost-effective.

While the original solar solution demonstrated how the system could be deployed autonomously in the field, the current AC-powered configuration allows us to focus resources on developing and testing the RF detection and SDR components without overextending the project's budget. In future phases, if funding or deployment requirements shift back toward remote, long-term operation, the previously designed solar-based solution can be revisited and integrated seamlessly into the system.

3.3 Communication Technologies

3.3.1. Broadcast Methods

3.3.1.1 Zigbee

Zigbee is a wireless communication protocol specifically designed for low-power, short-range applications such as sensor networks, automation systems, and smart home devices. It is built on the IEEE 802.15.4 standard and typically operates in the 2.4 GHz frequency band, although it can also operate on sub-GHz frequencies depending on the region and application. One of Zigbee's defining characteristics is its support for mesh networking, which allows devices within the network to communicate with each other directly or through intermediary devices. This feature increases the reliability and resilience of the network by enabling data to be rerouted around nodes that go offline or experience interference.

One of the biggest advantages of Zigbee is its extremely low power consumption, which makes it ideal for battery-powered devices that need to run for months or even years without being recharged or replaced. Zigbee modules can be put into deep sleep modes between transmissions, consuming only microamps of current when idle. This is why Zigbee is commonly used in applications such as smart thermostats, motion sensors, and remote-controlled lighting systems. Another key benefit is scalability—Zigbee networks can support hundreds of devices, and the mesh topology ensures that communication remains stable even as the network grows. These

features make Zigbee particularly attractive for projects that require a large number of interconnected nodes operating on limited power.

Despite these strengths, Zigbee also comes with some limitations that impact its usefulness in more complex or data-heavy systems—such as ours. For starters, Zigbee has a relatively **low** maximum data rate of 250 kbps. While this is sufficient for basic data logging or status updates (like a sensor reporting temperature or battery level), it is far too limited for our application, which may involve the transmission of RF signal characteristics, timestamps, GPS data, and other detailed logs from each detection node. This kind of real-time communication demands significantly more bandwidth than Zigbee can offer.

In addition to bandwidth limitations, Zigbee's effective range is also fairly limited, with typical distances reaching about 100 meters in open space. This range can drop significantly in the presence of physical obstructions or interference from other devices operating in the 2.4 GHz band. For a project like ours—which is designed for remote outdoor deployment across large open regions—this lack of range becomes a major challenge. We would need to add extra relay nodes just to cover longer distances, which adds cost, complexity, and potential failure points to the system.

That being said, Zigbee has been explored in aviation and UAV-related applications, although primarily in small-scale or internal systems. One study published by TheIRED (2023) looked into Zigbee's use in aircraft hydraulic monitoring systems. In this context, Zigbee was found to be a good fit because the data being transmitted was low-bandwidth, and the power-saving features helped reduce the load on the aircraft's electrical systems. Another study shared on ResearchGate (2023) examined Zigbee in multi-UAV coordination. It showed that Zigbee could be useful for basic communication between drones flying in close formation, though it clearly struggled when applied to high-speed data sharing or long-range control scenarios.

For our project—designing a passive aircraft detection system for remote, off-grid environments—Zigbee unfortunately does not meet the requirements needed for the core communication layer. Our system will eventually involve multiple detection units spaced over a wide geographic area, each potentially transmitting detection data in real time to a base station or cloud platform. These communications demand not only longer range but also more bandwidth than Zigbee is capable of providing.

However, that does not mean Zigbee is completely irrelevant to our project. While it won't serve as the main communication protocol between detection units or to the base station, it could still have value in localized applications within each unit. For example, Zigbee could be used for internal communication between subsystems, or for low-power environmental monitoring modules that report temperature, humidity, or battery status to the main processor. In that way, Zigbee could serve as a supporting protocol that enables more intelligent, energy-aware operation without adding much complexity or power drain.

Additionally, if future iterations of our project include smart diagnostics or automated maintenance systems, Zigbee could potentially be used to communicate between nearby nodes during field servicing. A technician might interact with the node wirelessly using a

Zigbee-compatible handheld device or tablet, accessing internal logs or configuring the system without needing a physical connection.

In conclusion, Zigbee is a proven and reliable technology in many low-power, low-data applications. Its strength lies in its simplicity, energy efficiency, and ability to form mesh networks that enhance reliability. However, its limited range and data rate prevent it from being the backbone of our aircraft detection system. That said, it could still play a role in smaller, less bandwidth-intensive tasks inside the unit. As we continue to develop and refine our system, we will keep Zigbee in mind as a secondary tool that could contribute to the internal functionality and energy savings of the overall platform.

3.3.1.2 Z-Wave

Z-Wave is a low-power wireless communication protocol designed primarily for smart home automation and Internet of Things (IoT) applications. It is commonly used to connect devices such as smart locks, motion detectors, light switches, and security systems, allowing them to operate in sync through a central hub or controller. Unlike many other short-range wireless protocols that operate in the crowded 2.4 GHz band (such as Wi-Fi and Bluetooth), Z-Wave operates in the sub-1 GHz frequency range. This gives it the advantage of avoiding much of the interference that can affect other wireless systems, making it particularly reliable in environments with many connected devices.

One of Z-Wave's most well-known features is its mesh networking capability. In a mesh network, devices do not need to communicate directly with the central hub. Instead, each device can pass data to its neighbors, creating multiple possible paths for the data to reach its destination. This means that if one device goes offline or a signal is blocked, the data can simply be routed through another path. This mesh functionality significantly improves the reliability and range of Z-Wave networks, especially in buildings where walls and furniture can obstruct signals. It also makes Z-Wave an appealing option in smart home setups, where many small devices are spread throughout a space.

Z-Wave is also valued for its energy efficiency, which is one of the main reasons it has been widely adopted for battery-powered devices. Z-Wave modules can run on very little power and remain in sleep mode most of the time, waking only when they need to send or receive data. This allows devices to operate for months or even years on a single battery, depending on usage. Another benefit is its strong interoperability. Z-Wave has a well-established standard, and most Z-Wave-certified devices from different manufacturers can communicate with one another without issue. This makes it a highly flexible choice for consumers and developers alike.

Despite all these positives, Z-Wave has some serious limitations—especially when it comes to systems like ours. The most significant of these is its limited data rate. Z-Wave can only transmit data at speeds of up to 100 kbps, which is enough for small commands like “turn on light” or “door is locked,” but completely insufficient for real-time transmission of complex RF detection data. Our aircraft detection system will need to send signal strength, time stamps, GPS coordinates, and other real-time data to a central server. That kind of data stream requires much faster transmission rates, and Z-Wave simply can't keep up.

Another issue is Z-Wave's limited communication range. In optimal, open-air conditions, Z-Wave may reach up to around 100 meters, but in real-world scenarios—especially those involving trees, terrain, or other physical obstructions—the effective range is often much less. This becomes a major problem for our project, which involves remote deployment of multiple detection units spread over wide areas. We need a system that can handle long-range data transmission, and ideally, one that can do so reliably without relying on a dense network of repeaters. Relying on Z-Wave would force us to install far more infrastructure just to maintain network connectivity, which adds cost, complexity, and additional power needs.

While Z-Wave has found some niche uses in aviation and UAV environments, it is generally limited to non-critical, low-bandwidth monitoring tasks. For example, it has been used in some ground-based airport systems for reporting equipment status or controlling lighting and environmental systems. According to GAO Tek (2023), Z-Wave has shown potential in localized control scenarios where low power consumption and simple status updates are prioritized. However, it has not been shown to perform well in use cases that involve large data sets or time-sensitive communication. That makes it a poor fit for any system that requires real-time analysis, such as the kind our aircraft detection platform depends on.

That being said, we don't completely rule out the potential use of Z-Wave in certain internal or auxiliary roles within a detection node. For instance, it could potentially be used for internal sensor monitoring, such as checking internal temperature, battery level, or case tampering alerts. These tasks don't require high-speed data or long-distance range, and Z-Wave could be a good way to offload that kind of monitoring from the main processor. This kind of internal communication could also be useful during system setup or maintenance, particularly if we later develop a local Z-Wave app that can wirelessly interact with a detection unit at close range.

In conclusion, Z-Wave is a strong protocol for short-range, low-power communication in fixed installations like homes or office buildings. Its ability to create stable mesh networks and its low energy use are definitely strengths, and it has earned its place in the world of IoT and home automation. However, the demands of our passive aircraft detection system—particularly the need for real-time, wide-area communication with moderate to high data throughput—simply exceed what Z-Wave was built to handle. It may find limited use in local monitoring tasks within a detection node, but it is not suitable as the primary communication method between detection units or to the central base station.

3.3.1.3 Near Field Communication (NFC)

Near Field Communication (NFC) is a short-range wireless technology that allows devices to communicate when they are only a few centimeters apart. It operates at 13.56 MHz and offers low data transfer speeds, typically ranging from 106 to 424 kbps (Wikipedia, 2023). NFC is widely used in everyday consumer applications, such as contactless payments, keyless access, and data sharing between smartphones and other smart devices. Its simplicity and security make it ideal for personal transactions and short-distance communication tasks.

NFC's main advantages include its ease of use, as it works with just a simple tap, allowing for quick and secure interactions. Because of its extremely short operating range, it is less vulnerable to hacking or signal interception, making it a good fit for applications where secure,

device-to-device communication is needed. It also has very low power consumption, which is helpful in battery-operated devices, especially those requiring infrequent use.

However, NFC has critical limitations that make it impractical for use in our aircraft detection system. Its extremely short range—just a few centimeters—prevents it from supporting any kind of long-distance communication. Additionally, its data rate is far too low for the type of real-time signal processing and communication required for aircraft detection. These limitations mean that NFC cannot support the bandwidth or range needed for unit-to-unit data sharing or cloud synchronization in our system.

In the aviation industry, NFC has been used to improve the passenger experience rather than for detection or data exchange between systems. Some airlines utilize NFC-enabled boarding passes that allow passengers to check in by tapping their phones at airport security gates (Airport Technology, 2023). Airports also use NFC for mobile ticketing, enabling travelers to purchase and store tickets on their smartphones (ResearchGate, 2023). Additionally, NFC can assist with real-time notifications such as gate changes or delay alerts, helping streamline the travel experience (QR Lab, 2023).

In conclusion, while NFC is excellent for short-range, secure, low-power interactions, it is not suitable for the needs of our passive aircraft detection system. The limited communication range and low data throughput prevent it from being used for real-time RF data sharing or system-wide connectivity. As such, NFC is not a viable option for this project.

3.3.1.4 Bluetooth

Bluetooth is a short-range wireless communication technology commonly used to connect devices like headphones, smartwatches, speakers, and smartphones. It operates in the 2.4 GHz frequency range and typically allows wireless data transfer over distances of up to 10 meters. Over the years, Bluetooth has evolved significantly, with the introduction of Bluetooth Low Energy (BLE), which was specifically designed to consume less power, making it well-suited for battery-powered and wearable devices (Bluetooth, 2023).

One of Bluetooth's biggest advantages is its low power consumption, especially in BLE mode, allowing devices to remain connected without quickly draining the battery. It is also widely compatible across a broad range of consumer electronics, thanks to being a standardized protocol supported by virtually all major device manufacturers. In addition to this, Bluetooth is generally easy to use, with most devices offering quick and simple pairing procedures, making it user-friendly for consumers and developers alike.

However, Bluetooth has notable limitations that make it a poor choice for our aircraft detection system. Its limited range—generally no more than 10 meters—makes it unsuitable for long-distance or wide-area communication. This is especially problematic in our project, where detection units are likely to be spaced far apart. Furthermore, Bluetooth operates on the same 2.4 GHz band as Wi-Fi, which can cause interference and reduce reliability, particularly in environments with many competing wireless signals. These drawbacks limit Bluetooth's usefulness in applications requiring real-time, uninterrupted data transmission across multiple detection nodes.

Bluetooth is primarily used for enhancing passenger experience, rather than for system-critical operations like detection. For instance, United Airlines and other carriers have begun installing Bluetooth-enabled in-flight entertainment systems, allowing passengers to connect their own wireless headphones during flights (United Media Room, 2023). Additionally, some airports use Bluetooth beacons to help guide passengers through terminals with real-time navigation, baggage tracking, and flight updates (MokoBlue, 2023).

In conclusion, while Bluetooth is a convenient and energy-efficient communication tool for short-range, low-bandwidth applications, its limited range and vulnerability to interference make it a poor fit for our passive aircraft detection system. However, it could still play a supporting role in localized communication between nearby modules or user interfaces, provided the devices remain in close proximity within a controlled setting.

3.3.1.5 Wi-Fi

Wi-Fi is one of the most widely used wireless communication technologies in the world today. It has become the standard for providing internet access in homes, schools, offices, airports, and just about any place where a wireless connection is needed. Operating mainly in the 2.4 GHz and 5 GHz frequency bands, Wi-Fi offers relatively high-speed data transmission over short to medium distances. More recently, advanced versions such as Wi-Fi 6 and Wi-Fi 6E have been introduced to increase speed, reduce latency, and improve performance in crowded environments. Wi-Fi 7 is also on the horizon, promising even better speeds and network efficiency.

One of the major strengths of Wi-Fi is its fast data rate. Depending on the standard in use, Wi-Fi can support data transfer rates of several hundred megabits per second or even more. This makes it perfect for high-bandwidth applications like streaming high-definition videos, playing online games, downloading large files, and video conferencing. It's also very easy to set up and widely available, with built-in support in nearly all laptops, smartphones, tablets, smart TVs, and a wide range of IoT devices. This level of compatibility means it integrates well into existing infrastructure, which is one of the reasons why Wi-Fi is so common in both personal and commercial environments.

In addition to fast speeds, Wi-Fi's cable-free operation allows for flexible device placement. Unlike wired Ethernet, which requires physical cables and limits mobility, Wi-Fi gives users the ability to move freely while staying connected. This is especially useful in larger indoor spaces, such as offices or warehouses, where running cables might be expensive or impractical. For small-to-medium area coverage where bandwidth is more important than range, Wi-Fi is often the default solution due to its convenience and speed.

However, when looking at Wi-Fi through the lens of our senior design project—a passive aircraft detection system meant for wide-area, outdoor use—there are several drawbacks that make it an impractical choice as the main communication technology. One of the biggest issues is range. While Wi-Fi might offer up to 100 meters of coverage in ideal open-air conditions, this range drops significantly when there are obstructions such as buildings, walls, trees, or uneven terrain. Our system is designed for deployment across wide open areas, where detection nodes may need to be hundreds of meters or even kilometers apart. Using Wi-Fi would require the installation of

numerous repeaters or mesh nodes just to maintain a stable connection, which adds cost, complexity, and power requirements that aren't ideal for solar-powered, remote installations.

Another major limitation of Wi-Fi is the potential for interference. Wi-Fi operates in frequency bands that are heavily used by other devices, including Bluetooth accessories, microwave ovens, baby monitors, cordless phones, and other Wi-Fi networks. This congestion can cause signal instability, dropped connections, or reduced speeds—especially in urban or populated areas. For a system like ours, which may need to transmit sensitive RF detection data in real-time, these types of interruptions are unacceptable. Reliable communication is critical, and Wi-Fi's vulnerability to interference introduces a risk we simply can't afford.

It's also important to consider the power requirements of Wi-Fi. While modern modules are more power-efficient than older ones, Wi-Fi is still not considered a low-power communication technology, especially when compared to alternatives like LoRa, Zigbee, or even Starlink SATCOM in standby mode. For a field-deployable system powered by solar panels and batteries, every milliamp of current matters. Using a Wi-Fi module that consumes power continuously, even when not transmitting, would drain the battery faster and reduce uptime, especially during periods of limited sunlight.

Wi-Fi is indeed used in a number of ways—but mostly for passenger experience and convenience, not for operational or mission-critical systems. For example, many airlines offer in-flight Wi-Fi, allowing passengers to access the internet during flights. This service is often limited in bandwidth, with reduced speeds and higher latency compared to home networks. Wi-Fi is also used in airports to provide public internet access, update flight information, and help travelers navigate terminals using location-aware services. Some baggage handling systems also use Wi-Fi for tracking and routing purposes. However, all these applications are relatively low-risk and do not require the kind of guaranteed real-time communication that our system demands.

In conclusion, while Wi-Fi is a well-known, fast, and flexible wireless communication method, it falls short in several key areas that matter for our project. Wi-Fi is excellent for indoor, consumer-level communication and works well for non-critical tasks like browsing or streaming. For our purposes, Wi-Fi is ideal for lab or small-scale outdoor tests where short-range, high-bandwidth signal delivery is needed. It could serve as the primary broadcast method for sending test signals to all SDR antennas simultaneously during controlled experiments.

3.3.1.6 LoRa

LoRa (Long Range) is a low-power wireless communication protocol developed by Semtech that is ideal for long-distance, low-data-rate applications. In the context of a radio diversity-based passive aircraft detection system, LoRa provides a practical solution for transmitting signal quality data—such as RSSI (Received Signal Strength Indicator), SNR (Signal-to-Noise Ratio), and time-of-arrival—from multiple RF sensor nodes to a central decision-making unit. This method is essential for implementing spatial radio diversity, where multiple nodes in different physical locations receive the same RF signal and relay their respective signal metrics. These

metrics are then compared in real time to determine which node has the best reception for further processing and data forwarding.

LoRa operates on unlicensed ISM frequency bands, such as 433 MHz, 868 MHz, and 915 MHz, depending on the region. Its key advantage lies in its chirp spread spectrum modulation, which allows signals to travel over long distances—even through obstacles—while maintaining energy efficiency. The system topology typically follows a star configuration, where individual receiver nodes send data to a central gateway equipped with a LoRa transceiver (e.g., SX1276). That gateway—typically a Raspberry Pi or similar SBC—collects data from all active nodes and uses software logic to identify the strongest and most stable signal path. This real-time comparison enables the dynamic selection of the best reception node, a cornerstone of the radio diversity technique.

From a power and cost standpoint, LoRa offers additional benefits. Its extremely low power consumption allows RF receiver nodes to operate on small solar panels and battery setups, extending their lifespan and reducing maintenance costs. Despite its strengths, LoRa does have limitations. Its maximum data rate is relatively low—around 50 kbps—so it cannot support high-bandwidth requirements like video streaming or raw RF sample transmission. However, for your system’s needs—transmitting metadata such as signal strength, timing, and simple flags—LoRa’s bandwidth is more than sufficient. Also, due to regional duty cycle restrictions in ISM bands, each node must be configured to avoid excessive transmission that may violate legal limits. These challenges are manageable with proper software throttling and timing mechanisms.

In conclusion, LoRa is an ideal communication method for your radio diversity-based system. It offers long-range, low-power communication over unlicensed spectrum, supports robust and secure data transmission, and is affordable for large-scale sensor deployment. It provides a robust method for status reporting and coordination in multi-node experiments. Nodes can report reception quality or confirm successful capture of test signals, enabling real-time monitoring of diversity performance. Its chirp-spread-spectrum modulation provides resilience against interference, making LoRa useful in challenging RF environments where lab or field conditions may not be ideal.

3.3.2 Integration

Integrating multiple SDR units into a single Raspberry Pi for broadcast experiments requires careful consideration of power delivery, data bandwidth, and synchronization. In our testbed, three RTL-SDR v3 units are connected via a powered USB hub to the Raspberry Pi. This configuration ensures that each SDR receives sufficient current without overloading the Pi’s onboard USB ports. The hub also simplifies cable management and allows the SDRs to be physically separated from the Pi, which can be advantageous for minimizing RF interference or positioning antennas in different locations.

The Raspberry Pi acts as both the central control unit and the data collector. Control scripts running on the Pi coordinate the start and stop of signal acquisition across all SDRs, ensuring that each device samples the test broadcast at the same time. This synchronization is essential for evaluating radio diversity effects, as precise timing allows the system to compare received signal strength, noise characteristics, and any multipath effects across nodes. Software libraries such as `pyrtlsdr` provide the interface for controlling each SDR individually, while Python scripts handle time-stamping and logging, enabling reproducible experiments.

Additionally, the Pi can interface with broadcast control protocols such as Wi-Fi or BLE to manage test sequences. For example, in a lab setting, a technician could trigger a broadcast over Wi-Fi while simultaneously logging SDR reception metadata. Low-bandwidth protocols like LoRa or Zigbee may also be integrated for node health monitoring or emergency stop commands. This hybrid approach ensures that while high-fidelity signals are sent over Wi-Fi or directly through signal generators, the Raspberry Pi maintains oversight of each SDR node's status and operational integrity.

Finally, considerations such as thermal management and RF shielding are important. When multiple SDRs are operated in close proximity, heat buildup can affect performance, and RF interference between units can distort received signals. Our testbed employs spacing and lightweight enclosures to mitigate these effects, ensuring that the experimental results reflect true diversity performance rather than artifacts of hardware layout.

3.3.3 Final Broadcast Selection

After evaluating a wide range of wireless communication technologies—including Zigbee, Z-Wave, NFC, Bluetooth, Wi-Fi, and LoRa—it became clear that none of these protocols alone could meet the unique requirements of our passive aircraft detection platform. Our system demands two distinct communication layers: (1) a high-fidelity broadcast mechanism for transmitting controlled test signals, and (2) a reliable digital channel for node-to-node coordination and data sharing. Based on these constraints, we selected an analog FM transmitter as the primary method for broadcasting messages during controlled experiments. FM provides a simple, robust, and interference-resistant means of transmitting a known reference signal that can be simultaneously received by all SDR units. This approach avoids the bandwidth limitations and range restrictions found in short-range digital protocols while giving us a clean signal for evaluating radio-diversity performance.

For inter-node communication, we adopt a TCP-based wireless networking layer between Raspberry Pi units. TCP provides guaranteed delivery, ordered packets, and reliability—features needed for transmitting metadata such as timestamps, SNR/RSSI reports, and capture confirmations across the system. This choice also aligns with our goal of modularity: the TCP link can operate over any underlying IP-capable medium (Wi-Fi, Ethernet, or future SATCOM integration), giving the system flexibility as deployment environments evolve. Together, analog

FM broadcasting for signal injection and TCP/IP communication for digital coordination create a hybrid architecture that meets the technical demands of our system while remaining scalable, maintainable, and robust for both laboratory testing and future field deployments.

3.4 Software Technologies

A multi-receiver radio diversity system requires software that can perform real-time signal processing, maintain stable wireless communication, synchronize audio streams, and make rapid switching decisions based on SNR/RSSI values. Selecting the correct development language is therefore critical to achieving reliability, low latency, and overall system performance. To evaluate the best language for both the Raspberry Pi receiver units and the Main Unit backend, we compared four commonly used high-level languages: MATLAB, Node.js, Java, and Python. Each language was analyzed according to its signal processing capabilities, SDR hardware compatibility, networking strength, multithreading performance, and suitability for user-interface development. This section presents a detailed evaluation of each language and concludes with a justification for selecting Python as the core technology for our Radio Diversity System.

Feature / Capability	Python	MATLAB	Node.js	Java
Real-Time Signal Processing (FM demod, filters, FFT)	Excellent — NumPy/SciPy, pyrtlsdr, fast enough for real-time	Great DSP tools but not real-time without Simulink	Not suitable for DSP	Moderate, but too heavy for real-time SDR
Hardware / SDR Integration	Native RTL-SDR support, GPIO/I2C/SPI on Raspberry Pi	Poor hardware control	Weak hardware access	Possible but complicated
Wireless Streaming (TCP/UDP)	Simple & reliable; ideal for SDR → central Pi streaming	Difficult	Good for web APIs, not IQ/audio streaming	Capable but verbose
Multithreading / Concurrency	Perfect for audio pipeline, jitter buffers, diversity logic	Limited	Strong async but not for DSP workloads	Strong but heavy to implement
GUI + Control Interface	Tkinter/PyQt — great for live SDR telemetry display	Decent (App Designer)	Must build as a web-app (overkill)	JavaFX works but heavy and slow

3.4.1. MATLAB

MATLAB is widely regarded as one of the best tools for digital signal processing, offering built-in functions for FFTs, filtering, time-domain visualization, and algorithm prototyping. Its graphical environment makes it ideal for academic research, simulation work, and offline RF analysis. However, MATLAB is not designed for embedded, real-time SDR deployment. It lacks native support for RTL-SDR hardware on the Raspberry Pi and cannot reliably handle high-rate

IQ streams or continuous FM demodulation without specialized toolboxes such as Simulink Real-Time, which require dedicated hardware and licenses.

MATLAB's networking capabilities are limited and overly abstract for real-time TCP/UDP streaming, making it difficult to support multiple simultaneous SDR audio feeds. Its parallel computing features also lack the fine-grained control needed for live jitter buffer management or multithreaded audio routing. While MATLAB is excellent for initial prototyping and visualization, it performs poorly as an embedded runtime environment and cannot meet the real-time responsiveness and hardware connectivity required by our diversity system. For these reasons, MATLAB was quickly ruled out as a deployment language and is more suitable for off-board analysis rather than active SDR processing.

3.4.2 Node.js

Node.js excels at asynchronous web applications and network servers, leveraging JavaScript's non-blocking event loop to handle thousands of HTTP requests efficiently. However, these strengths do not translate well to digital signal processing or radio system workloads. Node.js lacks optimized numerical computing libraries and cannot perform FM demodulation, spectral processing, or high-rate buffer manipulation at the speed required for SDR applications.

Hardware integration in Node.js is also limited. Accessing devices such as RTL-SDR receivers, sound cards, or GPIO/SPI interfaces requires third-party bindings that are slow, incomplete, or unreliable for real-time use. Although Node.js can handle TCP/UDP communication, it is ideal only for lightweight web APIs—not for streaming IQ or audio data.

Node.js could theoretically be used to build a cloud dashboard or web-based UI, but deploying it as the core radio processing engine would introduce severe performance bottlenecks. It is fundamentally unsuitable for real-time SDR, RF signal handling, or multi-receiver audio synchronization, making it a poor fit for this project.

3.4.3 Java

Java plays a supporting role in our system by powering the graphical user interface. With tools such as JavaFX or Android Studio, Java can efficiently display real-time telemetry, system mode states, and SDR connectivity status. Java's strong concurrency tools—threads, executors, synchronized queues—enable the UI to process incoming data while remaining responsive to user input. Its platform independence also makes the UI easy to deploy on laptops, tablets, and field devices.

However, Java is not well-suited for the real-time SDR backend. The JVM introduces unpredictable garbage-collection pauses, making precise timing difficult. Native hardware access is complex and often requires JNI bindings, which are cumbersome and error-prone. While Java can technically handle TCP/UDP streaming, it is far more verbose and difficult to debug

compared to Python’s networking tools. Its DSP performance is also significantly weaker, lacking the scientific ecosystem required for efficient signal processing.

Java is excellent for the user-facing control interface, but it lacks the performance and simplicity needed for real-time RF acquisition, making it complementary—but not central—to the system.

3.4.4 Python

Python provides the ideal foundation for both the Raspberry Pi receiver units and the Main Unit backend. Its scientific computing ecosystem, including NumPy, SciPy, pyrtlsdr, sounddevice, and socket, is optimized for exactly the type of SDR processing required in this project. With pyrtlsdr’s integration into the C-based librtlsdr driver, Python can stream IQ samples directly from RTL-SDR hardware and perform FM demodulation at real-time sample rates. These capabilities allow each SDR receiver to compute SNR/RSSI values, transmit audio, and send telemetry with minimal overhead.

Python’s threading and concurrency tools enable simultaneous processing of audio streams, telemetry packets, control messages, and data buffering—core requirements for stable SDR switching. The language’s readability and rapid development cycle allowed us to iterate quickly on buffer architecture, diversity logic, and latency fixes. Python’s socket library also made TCP streaming easy to implement, providing reliable and efficient real-time communication between the receivers and the Main Unit.

Moreover, Python is easy to debug, maintain, and extend. Whether adding new SDR units, introducing advanced diversity algorithms, or integrating cloud monitoring, Python offers unmatched flexibility. It is the only language in our comparison capable of handling real-time SDR processing, multi-threaded audio routing, and rapid prototyping in a unified environment.

3.4.5 Language selection

After evaluating MATLAB, Node.js, Java, and Python, it is clear that Python is the only language that meets all functional, performance, and development requirements of our Radio Diversity System. MATLAB provides excellent DSP tools but is impractical for embedded real-time operation. Node.js is excellent for web services but lacks the computational capabilities and hardware access needed for SDR processing. Java is ideal for building the user interface but too heavy for the timing-sensitive SDR backend.

Python, in contrast, offers the perfect balance of rapid development, real-time capability, SDR hardware compatibility, multithreading support, and networking reliability. Its extensive scientific ecosystem allows efficient FM demodulation and telemetry generation, while its clean syntax and modularity streamline debugging and refinement during development. The availability of optimized C-backed libraries ensures that Python can achieve the performance

required on Raspberry Pi hardware, and its flexibility allows future expansion of diversity logic and system capabilities.

For these reasons, Python was selected as the core programming language for our radio diversity project. It enables robust real-time performance, simplifies distributed system design, and provides a sustainable foundation for future engineering work.

3.5 Development Environment

Selecting an appropriate development environment is important for ensuring efficient coding, debugging, and deployment of software across both the Raspberry Pi SDR units and the Main Unit backend. Each tool offers different advantages depending on the stage of development—rapid prototyping, debugging, visualization, or system integration. The following sections provide a detailed analysis of the environments listed in the table, explaining their strengths, limitations, and relevance to our Radio Diversity System.

Environment / Tool	Strengths	Limitations
Thonny	Pre-installed on Raspberry Pi, easy to use, beginner-friendly.	Limited advanced features and Git integration.
PyCharm	Powerful IDE with great debugging and project tools.	Heavy on resources, slower on Pi.
Jupyter Notebook	Great for visualization and quick testing.	Poor for multi-file or structured projects.
Raspberry Pi OS (with IDLE)	Lightweight, built-in editor.	Basic, lacks advanced debugging or project tools.
Anaconda (Spyder)	Rich scientific computing support.	Bulky install, not optimized for embedded use.

3.5.1 Thonny

Thonny is the default Python IDE pre-installed on Raspberry Pi OS, making it immediately accessible without requiring additional installation steps. Its clean interface and beginner-friendly layout make it ideal for quick script edits, small test programs, or debugging simple functions directly on the Raspberry Pi hardware. Because Thonny is lightweight, it loads quickly and runs smoothly even on the Pi's limited resources, which is useful when testing scripts related to SDR initialization, GPIO reads, or telemetry formatting.

However, Thonny lacks advanced features needed for large, multi-file system development. It has limited project management capabilities, minimal Git integration, and no support for complex refactoring or professional-level debugging tools. For a project like ours—which involves multiple Python modules, multi-threaded processes, and network servers—Thonny is

useful only for quick patching or small edits directly on the device, not for full-scale development.

3.5.2 PyCharm

PyCharm is one of the most powerful and feature-rich Python IDEs available. It offers intelligent code completion, sophisticated debugging, breakpoints, virtual environment management, deep project structuring, and built-in Git support. For the Radio Diversity System, PyCharm was highly valuable during the early development stages on the ASUS NUC and on the laptop-based backend testing environment. It allowed us to navigate complex module hierarchies, step through multi-threaded code execution, debug socket connections, and identify buffer-related issues.

The drawback of PyCharm is its resource usage. It requires significant CPU and RAM to run efficiently, making it less suitable for the Raspberry Pi. Running PyCharm directly on a Pi can lead to slow performance and long load times, especially during indexing or linting. Therefore, PyCharm is ideal for development on a full PC or the NUC—but not practical for on-device coding.

3.5.3 Jupyter Notebook

Jupyter Notebook is an exceptional tool for visualization, quick testing, and exploratory analysis. It allows developers to run code in small, interactive cells, which is extremely useful for plotting SNR/RSSI trends, testing buffer algorithms, analyzing recorded audio, or experimenting with mathematical transformations on sample SDR data. Jupyter was also helpful when evaluating the impact of jitter, comparing audio delays, or visualizing telemetry logs.

However, Jupyter Notebook is not suitable for developing or managing multi-file, production-level systems. It does not handle threaded programs well, cannot manage long-running servers reliably, and lacks the modular structure needed for the full Radio Diversity System. It is best used as a research tool rather than a deployment or integration environment.

3.5.4 Raspberry Pi OS (with IDLE)

Raspberry Pi OS includes IDLE, a very lightweight Python editor useful for making quick edits directly on the Pi. Its simplicity ensures compatibility even with limited system resources, and it can be helpful for testing small code segments or verifying that dependencies are installed correctly.

However, IDLE lacks modern development tools such as Git integration, advanced debugging, breakpoints across multiple files, project organization features, and terminal-based toolchains. It is useful only for basic testing or editing short scripts. For complex system components—such as

the multi-threaded audio server, telemetry handlers, or UI communication modules—IDLE is too limited and not adequate for full development.

3.5.5 Anaconda (Spyder)

Anaconda with the Spyder IDE provides a powerful scientific computing environment with extensive package management, integrated scientific libraries, and an interface similar to MATLAB. Spyder is well-suited for data analysis, offline simulations, and plotting. Its environment makes it easy to experiment with DSP operations, inspect recorded telemetry, and analyze audio snapshots generated by the system.

However, Anaconda is extremely bulky and not optimized for embedded platforms such as the Raspberry Pi. The environment consumes large amounts of disk space, has heavy startup overhead, and installs many packages unnecessary for real-time SDR operation. Running Spyder on a Pi is impractical due to performance constraints. Therefore, Anaconda is useful for offline testing on a laptop or desktop but unsuitable for deployment or live debugging on embedded hardware.

3.5.6 Why We Chose Thonny for Our Project

After evaluating multiple development environments, Thonny was selected as the primary tool for on-device development and testing in our Radio Diversity System. Although more advanced IDEs such as PyCharm and Spyder provide richer debugging and project management features, they are too resource-intensive for the Raspberry Pi, especially when running real-time SDR processes, multi-threaded audio streams, and continuous telemetry updates. Thonny's lightweight footprint makes it ideal for the constrained hardware of the Raspberry Pi, allowing our team to edit, test, and debug code directly on the device without significantly impacting CPU or memory usage.

Thonny is also pre-installed on Raspberry Pi OS, which reduces setup time and avoids additional dependency issues. Its simple interface allows us to quickly open the audio server, telemetry handler, or control logic files and make rapid modifications during field testing—something that was essential when diagnosing latency problems, buffer synchronization issues, or SDR connection timeouts. The ability to immediately run Python scripts on the same hardware they were deployed on improved accuracy during troubleshooting and prevented environment-related inconsistencies.

Although Thonny lacks advanced Git integration, profiling tools, and multi-file project structuring, its strengths aligned perfectly with the core needs of our project: reliability, accessibility, and low overhead. Most large-scale development was completed on the ASUS NUC using PyCharm, while Thonny served as the ideal environment for live testing, quick fixes, and validation on each Raspberry Pi. Ultimately, Thonny provided the most stable and efficient

on-device development experience, making it the best fit for our embedded real-time SDR system.

3.6 Data Transfer Mechanism

The data mechanism in our Radio Diversity System is designed to reliably transport real-time audio, telemetry, and control information between multiple distributed SDR receiver units and the central processing node. Because the system must operate continuously and make fast diversity decisions based on RF signal conditions, the underlying data flow needs to be efficient, synchronized, and tolerant to delays or temporary disconnections. At its core, the data mechanism combines structured TCP communication, multi-threaded buffering, and selective audio routing to ensure that the strongest and cleanest signal is always forwarded to the audio output while maintaining complete visibility of each receiver's status.

Each SDR receiver node sends two primary data streams to the Main Unit: telemetry and audio. Telemetry packets include SNR, RSSI, frequency, connection status, and sequence timestamps. These packets are transmitted over a lightweight TCP connection, which ensures ordered delivery and reliability so that the Main Unit always has an up-to-date picture of each receiver's signal environment. Since telemetry data is small in size and low-frequency, the system can update SNR and RSSI indicators nearly instantaneously, allowing the UI and Mode Selection logic to react immediately when RF conditions change. This fast telemetry channel forms the foundation of the diversity algorithm, enabling intelligent switching between receivers.

The audio data mechanism is more complex because each SDR streams continuous audio samples at 48 kHz. To handle this high-throughput stream without loss, each receiver uses a dedicated TCP audio socket. The Main Unit unpacks each audio frame using a structured header containing a magic word, sequence number, sample count, and sampling rate. These frames are pushed into per-receiver audio queues that buffer incoming audio. Sequence numbers enable the Main Unit to reorder packets, reconstruct continuous audio, and populate per-receiver jitter buffers before forwarding audio to the output. This mechanism ensures smooth playback even when packets arrive slightly late or out of order.

Once audio is buffered, the Main Unit uses a selection mechanism to determine which receiver's audio should be played. In Manual mode, only the chosen receiver's buffer is used. In Threshold mode, the system prioritizes SDR1 as long as its SNR stays above the configured threshold; otherwise, it switches to the strongest alternative. In Auto mode, the system continuously compares SNR values across all receivers and selects the one with the highest signal quality. These decisions occur at runtime and are tightly coupled with the telemetry stream, ensuring that audio selection responds immediately to RF environment changes.

To maintain system stability and low latency, the data mechanism includes logic for disconnect detection, buffer clearing, and stale-audio prevention. If an SDR stops transmitting or loses connection, the Main Unit identifies the timeout and immediately marks that receiver as disconnected, clears its buffer, and prevents stale audio from being selected. Similarly, when switching from one receiver to another, the system clears the newly focused SDR's playback buffer so it aligns with real-time audio instead of outputting old or delayed samples. This

prevents compounding delays and ensures that diversity switching does not introduce additional latency into the audio path.

Finally, all processed telemetry and selection decisions are packaged and transmitted to the User Interface through a dedicated UI socket. This allows the UI to display real-time SNR, RSSI, connection status, and currently focused receiver. At the same time, the UI can send control commands back to the Main Unit—such as changing frequency or adjusting selection mode—through the command queue. In combination, these data flows create a complete, synchronized, and fault-tolerant communication architecture that allows the system to perform real-time radio diversity processing.

3.7 Data Transmission Process

The Data Transmission Process is the core operational pathway that enables audio streams, telemetry updates, and control messages to flow reliably between the distributed SDR receiver nodes and the central processing unit. Because the Radio Diversity System must detect RF signals in real time and switch between receivers based on instantaneous signal quality, the transmission process must be both fast and dependable. This system relies primarily on TCP-based communication channels, which provide ordered, lossless packet delivery—ensuring that audio samples and telemetry packets arrive intact and in sequence.

Each SDR node initiates two separate outbound TCP connections when it boots: one for audio data and another for telemetry data. These independent channels prevent high-volume audio traffic from interfering with critical low-latency telemetry updates. The telemetry channel transmits JSON-formatted messages containing real-time SNR, RSSI, receiver status, and current tuned frequency. Because these packets are lightweight and transmitted every few milliseconds, they provide the Main Unit with a continuous view of RF conditions across all receivers. This fast telemetry layer is essential for accurate mode selection, link switching, and UI updates.

The audio transmission process is more bandwidth-intensive and therefore requires a structured and highly efficient mechanism. The SDR converts raw RF signals into demodulated audio samples at 48 kHz, packages these samples into binary frames, and attaches a header containing a magic identifier, sequence number, and sample count. Once transmitted over TCP, the Main Unit receives and reconstructs the stream based on these sequence numbers. This guarantees that audio chunks are assembled in the correct order, even if packets arrive out of sequence due to network jitter or interference. The Main Unit then places the audio frames into per-receiver jitter buffers, which smooth out transmission timing variations and provide stable playback during diversity switching.

In addition to telemetry and audio, a third communication path exists for control transmission, where the Main Unit sends commands back to the SDRs. These commands include frequency changes, restart instructions, or configuration updates. The control channel allows operators to tune receivers wirelessly without physical access to each node. All commands are formatted as JSON instructions and transmitted over persistent TCP sockets with automatic reconnection logic, ensuring reliability even in unstable wireless environments.

A crucial part of the Data Transmission Process is connection monitoring and timeout handling. The system continuously tracks the last time each SDR sent a packet. If telemetry or audio data stops arriving within a specified timeout window, the Main Unit marks that SDR as disconnected, clears its audio buffers, and prevents stale audio from being selected. This ensures that the system never routes outdated or delayed audio to the output—an essential requirement for maintaining real-time performance. Once the SDR reconnects, the transmission process automatically reinitializes without requiring a manual restart.

Finally, all processed data—including selected audio output, mode state, and telemetry summaries—is forwarded to the User Interface transmission pipeline, which updates the operator dashboard. Through this interface, users can observe signal strength in real time, verify which SDR is currently selected, and send control actions back to the system. This closed-loop transmission cycle—SDR → Main Unit → UI → Main Unit → SDR—creates a fully synchronized and fault-tolerant communication architecture that supports continuous real-time RF monitoring and dynamic diversity switching.

3.8 Data Structure and Storage

The Data Structure and Storage subsystem defines how information is organized, buffered, and preserved throughout the Radio Diversity System. Because the system continuously receives high-frequency audio streams and rapidly updating telemetry from multiple distributed SDR receivers, it requires a storage architecture that is efficient, structured, and capable of supporting real-time operations without introducing delays. The design focuses on fast in-memory buffering for live processing and lightweight file-based storage for user-initiated snapshots and debugging.

At the core of the system's data organization are structured telemetry objects and binary audio frames, each optimized for its respective workload. Telemetry from each SDR is transmitted using compact JSON messages that contain SNR, RSSI, frequency, timestamps, and connection state. This format is deliberately chosen for its readability, cross-platform compatibility, and minimal parsing overhead. The Main Unit stores these telemetry values in a standardized in-memory dictionary, ensuring that the latest measurements are always available for diversity decision-making, disconnect detection, and UI updates. By overwriting old values with new ones, the system avoids unnecessary memory growth while maintaining an up-to-date snapshot of network conditions.

Audio data, which requires far higher throughput, is managed using a structured binary data format. Each SDR sends audio frames containing a header with a magic identifier, sequence number, sample count, and sampling rate, followed by raw PCM audio samples. This binary structure ensures efficient transmission while giving the Main Unit the metadata needed to reorder frames, detect missing packets, and reconstruct uninterrupted audio streams. Sequence numbers play a critical role in maintaining audio continuity across receivers, especially during periods of network jitter or when audio arrives out of order.

To support real-time playback, the system relies on a hierarchy of in-memory buffers and queues, designed to hold just enough data to smooth out transmission irregularities without accumulating excessive latency. Each SDR maintains its own audio queue for incoming frames, a reassembly buffer indexed by sequence number, and a playback deque that feeds the audio output stream. Similar buffer structures maintain short-term audio history for each receiver, capturing several seconds of recently played audio. This history enables features such as saving snapshots without pausing the live stream. By limiting buffer sizes and dropping the oldest data when full, the system prevents lag buildup and ensures that audio remains tightly synchronized with real-time RF conditions.

For persistent storage, the system uses on-demand file generation, primarily when a user chooses to save a snapshot. When triggered, the system concatenates recent audio chunks from the selected SDR's history buffer and writes them into a WAV file using standard PCM formatting. This approach ensures high-fidelity reproduction of the audio and maintains compatibility with external analysis tools. The use of short, compressed metadata in telemetry logs and structured chunk-based WAV storage keeps disk usage minimal while still providing valuable debugging information.

Finally, optional event logging stores disconnects, mode changes, SNR drops, and other operational events in simple timestamped text files or JSON logs. These persistent logs provide traceability and support long-term system evaluation—helping developers diagnose issues such as wireless interference, SDR instability, or selection-mode misbehavior. Together, these storage mechanisms form a balanced architecture: fast enough for real-time processing during operation, yet structured enough to support historical analysis, debugging, and validation.

Chapter 4 Standards and Design Constraints

4.1 Software

4.1.1 Software Standards

To ensure the reliability, maintainability, and long-term performance of our Radio Diversity System, it is important to follow recognized industry software standards. These standards guide how software is designed, implemented, tested, and maintained, and help ensure that the system remains consistent, secure, and scalable as new features or hardware changes are introduced. Two of the most relevant standards for our project are IEEE 12207, which governs the full software development lifecycle, and ISO/IEC 27001, which provides guidelines for secure handling of system data and network communication.

4.1.1.1 IEEE 12207 – Software Life Cycle Processes

The IEEE 12207 standard provides a structured framework for software development and maintenance, ensuring that all stages of the software lifecycle are well-defined, repeatable, and scalable. This standard is highly applicable to our Radio Diversity System because our software

must handle continuous updates, real-time audio streaming, multi-threaded signal processing, and network communication across multiple SDR receivers. Since the system must remain stable while processing real-time RF signals, managing multiple TCP connections, and switching between receivers with minimal latency, adhering to a formalized software methodology is essential for ensuring stability and overall system reliability.

A key strength of IEEE 12207 is its systematic approach to requirements analysis. In our project, both functional and non-functional requirements needed to be clearly defined before development began. Functional requirements included the ability to demodulate FM audio in real time on each Raspberry Pi, extract and transmit SNR/RSSI telemetry, support Manual, Threshold, and Auto diversity modes, and maintain synchronized communication with the Main Unit and User Interface. Non-functional requirements focused on achieving low-latency performance, maintaining consistent wireless communication, avoiding software-induced audio delay, and ensuring the software remained maintainable for future enhancements. By defining these requirements early, we ensured that the system architecture could support real-time operation under various deployment scenarios.

Once requirements were established, the software design and development phase emphasized modularity and scalability. The architecture was divided into independent components—such as the audio receiver, telemetry server, control server, buffer manager, and diversity selection logic—so that each module could be tested individually and integrated without disrupting the broader system. This modular design follows IEEE 12207 best practices and allows updates to one part of the system (e.g., improving buffer logic or adding frequency control) without affecting the others. Additionally, object-oriented structuring, thread separation, and API-based communication help ensure that the system remains flexible and extensible.

Testing and validation represent another core requirement of IEEE 12207 and were instrumental in improving system performance. Unit tests validated the correctness of individual modules, such as demodulation routines, JSON telemetry parsing, and socket communication. Integration tests ensured that all components functioned cohesively across three SDR receivers and the Main Unit. A significant portion of testing was devoted to real-time validation—measuring audio latency, evaluating buffer behavior, stress-testing network conditions, and verifying correct receiver switching under changing SNR conditions. These tests were crucial in identifying the original large-latency issue caused by unbounded queues, which was resolved by implementing bounded buffers and trimming upon SDR switching.

After the system successfully passed testing, IEEE 12207 emphasizes deployment and long-term maintenance. In our project, deployment included configuring systemd services for automatic startup on the Raspberry Pi and Main Unit, implementing version control through Git repositories, and ensuring that the system could run autonomously for extended periods. Maintenance procedures involve monitoring logs, evaluating performance metrics, updating firmware or software modules, and applying security patches as needed. This structured deployment approach ensures that the system can evolve reliably as new requirements emerge.

Documentation and traceability also represent key pillars of IEEE 12207. Throughout development, we maintained comprehensive documentation, including software design diagrams, flowcharts, module descriptions, buffer logic explanations, and step-by-step debugging

logs. Proper documentation ensures that future team members or engineers can understand, extend, or modify the system without compromising functionality or introducing unnecessary complexity.

By following the IEEE 12207 standard, our software development process remained structured, scalable, and optimized for long-term reliability. This approach helped reduce development risks, improved team coordination, and ensured that the Radio Diversity System remained maintainable and adaptable as new features were introduced. Given the complexity of real-time multi-receiver audio processing, adhering to this internationally recognized standard significantly contributed to the success, robustness, and sustainability of our project.

4.1.1.2 ISO/IEC 27001 – Information Security Management

Because our Radio Diversity System relies on continuous wireless communication, TCP-based networking, telemetry transmission, and remote data visualization, ensuring information security and data integrity is essential. The ISO/IEC 27001 standard provides a globally recognized framework for establishing a secure information management process, helping protect system data, prevent unauthorized access, and maintain safe operation of distributed SDR receivers. This standard is highly relevant to our project since the system involves multiple Raspberry Pi units transmitting audio data and SNR/RSSI telemetry over WiFi networks to a centralized Main Unit, which introduces potential vulnerabilities such as packet interception, spoofing, or unauthorized access to control interfaces.

A fundamental principle of ISO/IEC 27001 is the protection of data confidentiality during transmission. Although the SDR audio streams themselves do not contain sensitive personal information, the system still transmits operational data—receiver identifiers, SNR values, device connection states, and backend control commands—that must be protected from manipulation or unauthorized monitoring. To address this, secure communication protocols such as Transport Layer Security (TLS) can be integrated into the TCP control and telemetry channels. TLS ensures that communication between the Raspberry Pi receivers and the Main Unit is encrypted, preventing eavesdropping or packet alteration. Furthermore, encrypted channels make it significantly more difficult for attackers to inject falsified telemetry values or disrupt diversity logic decision-making.

Another core element of ISO/IEC 27001 is strong access control. In our Radio Diversity System, access to system functions—such as changing operating modes, modifying frequency settings, or managing receiver connections—should be restricted to authorized operators only. To support this, we can implement role-based access control (RBAC) on the Main Unit UI and secure it behind user authentication. Only authenticated operators would be able to issue control commands such as `set_freq` or initiate snapshot recordings. Additionally, the internal control socket used by each Raspberry Pi to receive frequency updates must be protected against unauthorized external connections, for example by restricting IP access, requiring authentication tokens, or running communication within a private network segment.

ISO/IEC 27001 also emphasizes risk assessment and continuous monitoring. For our system, this involves evaluating potential cybersecurity risks related to WiFi instability, open ports, unsecured APIs, and long-duration autonomous operation. We conducted risk assessments to

identify vulnerabilities such as unencrypted telemetry, unsecured control ports, and the potential for buffer manipulation through malformed packets. By identifying these risks early, we applied mitigation strategies such as validating all incoming telemetry, sanitizing UI commands, and implementing disconnect and timeout logic to prevent rogue devices from affecting system behavior. Regular monitoring of logs—including connection attempts, error messages, and unexpected control input—helps detect suspicious activity and maintain system integrity.

Another important ISO/IEC 27001 measure is logging and auditability. Our system maintains detailed logs through the Main Unit, including receiver connection events, telemetry updates, mode switching activity, and buffer-related warnings. These logs serve as an audit trail, allowing developers or administrators to analyze unexpected behavior, identify possible tampering attempts, or diagnose system faults. Maintaining logs also supports long-term system evaluation, helping identify patterns such as repeated network dropouts, signal anomalies, or unexpected SDR behavior.

Compliance with ISO/IEC 27001 further ensures alignment with broader cybersecurity frameworks such as the NIST Cybersecurity Framework. Although our Radio Diversity System is primarily used for research and Senior Design demonstration purposes, following these standards helps ensure that the system can be safely deployed in more demanding environments or adapted into more advanced RF monitoring platforms in the future. If cloud interfaces or remote dashboards were added later, ISO 27001-compliant hosting—such as AWS, Azure, or Google Cloud—would be required to maintain consistent security practices.

By applying ISO/IEC 27001 principles to our Radio Diversity System, we establish a robust and proactive security foundation. This ensures that telemetry, control commands, and receiver connections remain protected from unauthorized access and cyber threats. Adhering to this standard improves data integrity, strengthens system resilience, and enhances operator trust, ensuring that the system performs safely and reliably even under wireless networking conditions or long-duration operation.

4.2 Hardware

Developing a multi-receiver SDR diversity system still comes with several hardware considerations, but the challenges are very different from the original aircraft detection idea. Since our system is now built around three RTL-SDR receivers, multiple Raspberry Pis, and a central processing unit, the main concerns are making sure each device receives strong RF signals, stays synchronized enough for meaningful comparisons, and can send audio and telemetry to the central Pi without dropping out. Even though our setup is smaller and simpler than the original plan, we still need to think carefully about hardware reliability, networking, and how each device interacts with the environment around it.

One of the biggest hardware-related concerns in our project is the performance of the RTL-SDR dongles. These low-cost receivers are easy to use, but they are sensitive to noise and can behave differently depending on USB power, antenna quality, or local interference. Because of that, we had to make sure each receiver was given a stable power source and a basic antenna that works consistently indoors. Our antennas do not need to be directional or wideband like the LPDA

antennas originally planned for aircraft detection; instead, simple whip or dipole antennas are enough for receiving an FM broadcast in our test environment.

Another important hardware challenge is ensuring that each Raspberry Pi can handle the tasks we ask of it—demodulating FM audio, gathering SNR or RSSI values, and streaming those to the central Pi. While the Raspberry Pi 4 is fairly capable, running multiple tasks at once can still push it if we're not careful. During development, we had to make sure cooling was adequate and that network throughput stayed consistent, especially when all three Pis were sending data at the same time. Fortunately, because our system works indoors and doesn't need to run on batteries or solar power, we don't have to worry about harsh outdoor environments or power-saving modes like before.

Networking is another key hardware constraint. The system works best when the network connection between the receivers and the central Pi is stable and low-latency. Wi-Fi works but can introduce jitter if the signal is weak or if multiple devices share the same access point. Ethernet is more reliable, so we often preferred using a switch when possible. The diversity logic depends on timely updates, so random network delays can affect how smooth the audio transitions are. Because of this, we spent time testing different network setups to make sure the hardware consistently supported real-time data streaming.

Finally, we also had to consider how the hardware integrates with the touchscreen interface. The central Raspberry Pi handles a lot at once: receiving audio, processing telemetry, running the diversity algorithm, and updating the touchscreen UI. Because the UI uses Tkinter and Matplotlib, both of which can be CPU-heavy if we're not careful, we designed the interface to be simple and lightweight. We avoided unnecessary animations or complicated graphics so the hardware wouldn't slow down during switching events.

In summary, the hardware requirements for our SDR Diversity System ended up being more about stability and consistency rather than environmental durability or long-range operation. By focusing on stable power, reliable antennas, and a clean network connection, we created a setup that performs well in real-time demonstrations and stays within the limits of what the Raspberry Pi and RTL-SDR hardware can handle.

4.2.1 Hardware Standards

When building our SDR Diversity System, we still needed to follow several hardware-related standards, even though our system is much simpler than the aircraft detection design we originally planned. Instead of dealing with long-range antennas, SATCOM terminals, or outdoor weatherproof nodes, our project focuses mostly on commercial electronics like Raspberry Pis, RTL-SDR dongles, and small antennas. Because of that, the hardware standards we follow are mostly related to consumer RF devices, USB-powered electronics, and basic safety practices for low-voltage systems.

One of the most important standards relevant to our design is FCC Part 15, which governs unlicensed radio receivers and low-power RF devices in the United States. Since RTL-SDR dongles and the simple antennas we use fall under this category, it's important that our system

operates only as a receiver and never transmits. Everything we built is strictly receive-only, so we comply with FCC rules by not generating or radiating RF signals.

Another standard that matters in our design is USB power safety, especially because all our RTL-SDR dongles rely on USB power from the Raspberry Pis. USB 2.0 and USB 3.0 both have defined current limits, and if a device draws too much power or gets too hot, it can damage the port or cause the Pi to throttle. RTL-SDR dongles are known to run warm during operation, so part of following hardware standards involves making sure the dongles have proper airflow and that we use reliable USB cables and power supplies. It's a small detail, but it helps keep the system stable.

Networking standards also play a role in our setup. Our system uses either IEEE 802.11 (Wi-Fi) or IEEE 802.3 (Ethernet) to stream audio and telemetry between the receivers and the central unit. These standards define the communication rates, timing, and expected behavior of the network. While we don't dive deeply into the protocol details, staying within these standard technologies ensures that our devices communicate reliably and stay compatible with typical routers and network switches.

For general electrical safety, our project also falls under low-voltage consumer electronics guidelines, which mostly include making sure power supplies are rated correctly, cables aren't overloaded, and components stay within their recommended operating temperatures. We aren't dealing with anything high voltage or industrial, so the risks are low, but it's still important that our hardware is arranged safely and runs within standard limits.

Overall, the hardware standards for our SDR Diversity System are more straightforward compared to the original aircraft-focused design. Most of them relate to everyday RF receivers, USB-powered peripherals, and common networking equipment. By staying within these established standards, our system remains safe, reliable, and compliant while keeping the design simple enough to develop and test as a student project.

4.2.2 Hardware Design Constraints

Even though our SDR Diversity System is much simpler than the original aircraft detection design, there are still several hardware constraints we had to work around. These constraints mainly come from the limitations of the RTL-SDR dongles, the processing ability of the Raspberry Pi units, and the need to keep all three receivers communicating smoothly with the central node. None of these constraints are extreme, but they did shape the way we designed and tested the system.

One of the biggest hardware limitations in our project comes from the RTL-SDR dongles themselves. These devices are great for low-cost RF work, but they are not professional-grade receivers. They are sensitive to noise, can overheat during long operation, and their signal quality depends heavily on USB power stability. Because of this, we had to make sure each Raspberry Pi used a reliable power supply and that the dongles had enough airflow during longer tests. We also noticed that different dongles sometimes behaved slightly differently, which meant we had to account for small variations in signal strength when comparing receivers.

Another important constraint involves the Raspberry Pi 4. While the Pi is capable of handling FM demodulation and telemetry extraction, doing this for long periods while also streaming data to the central unit can push its CPU usage higher than expected. This meant we needed to keep each receiver's workload as simple as possible. We avoided adding unnecessary processing steps or complicated filtering on the receiver side. Instead, the Pi focuses only on the essentials: tune, demodulate, extract SNR/RSSI, and stream. This kept the hardware operating smoothly without overheating or throttling.

Networking limitations also played a role in shaping our hardware decisions. Our system relies on a stable network connection to send audio and telemetry from each receiver to the central Pi. Wi-Fi is convenient but can introduce random spikes in latency, especially if the signal is weak or multiple devices share the same router. Ethernet is more reliable, but only works when the Pis are physically close enough to be wired together. Since the diversity logic depends on timely updates, any hardware-related network issue can cause the switching to lag or audio to stutter. This made us carefully consider test placement, router settings, and whether wireless or wired networking made more sense for each experiment.

The touchscreen interface also adds hardware constraints to the central Raspberry Pi. The Pi needs to handle both backend signal processing and frontend UI rendering at the same time. Tkinter and Matplotlib can be demanding if used carelessly, so we had to design the UI to be lightweight and avoid unnecessary graphical effects. Keeping the UI simple meant the Pi didn't slow down during switching events or while receiving telemetry from all three receivers.

Lastly, because our antennas are simple consumer whip antennas rather than large, directional, or outdoor-rated antennas, the placement of each receiver matters a lot. Indoors, signals can bounce around or get blocked easily, so the hardware constraint here is environmental: each antenna needs a clear enough spot to receive the FM broadcast reliably. This doesn't require professional mounting equipment, but it does require some awareness of how sensitive low-cost RF gear can be.

Overall, the hardware constraints in our SDR Diversity System were manageable, but they definitely guided our design choices. By understanding the limitations of the RTL-SDRs, Raspberry Pis, and home networking hardware, we were able to build a system that performs reliably within those boundaries. The constraints also helped keep the project realistic and grounded in what's achievable with student-level hardware.

4.2.3 Non-Technical Constraints

Even though our project is mostly centered around software-defined radio and signal processing, there are still a number of non-technical hardware constraints that affected the way we planned, built, and tested the system. These constraints aren't about RF theory or electronics, but more about practical limitations, budgeting, availability, and how much complexity we could realistically handle as a student team within the semester timeline.

One of the biggest non-technical constraints we faced was **cost**. The original aircraft detection concept required expensive hardware like high-gain antennas, specialized RF analyzers, and potentially satellite communication equipment. Once the project shifted toward an SDR diversity

system, we had to keep it affordable and stick with consumer-grade components. This is why we used RTL-SDR dongles and Raspberry Pi units instead of more advanced SDR platforms like USRPs or RadioVerse boards. These low-cost components made the project achievable, but also meant accepting limitations in performance and reliability.

Another constraint was availability of hardware. During Senior Design, certain components especially Raspberry Pis—are often out of stock or overpriced. Because of this, we had to work with the hardware we already had access to rather than designing around ideal components. The RTL-SDR dongles we used are easy to obtain, but their antennas and USB cables can vary in quality, which affected signal consistency. This meant we had to design around variability rather than assuming perfect hardware conditions.

We also had to consider time and complexity constraints. Building a synchronized multi-receiver RF system can become extremely complicated if you start adding features like phase alignment, TDOA localization, or advanced filtering. As a student team with limited time, we had to simplify the architecture to focus on one achievable goal: showing real-time diversity by selecting the cleanest signal. Keeping the hardware layout simple helped us spend more time on testing and debugging rather than trying to implement features that would have pushed the project beyond our realistic timeline.

Another non-technical constraint involved physical space and testing conditions. Since our system uses small antennas and indoor receivers, we had to work within the limits of the rooms, hallways, and lab spaces available on campus or at home. Indoor environments often cause reflections, multipath, and interference, which we had to account for during testing. While these factors are technically RF-related, the constraint itself is non-technical because it's tied to the facilities and spaces we had access to rather than the design of the hardware.

We also had to think about ease of use and portability, since the final system needed to be demonstrated clearly during presentations. The touchscreen interface, wiring layout, and power connections had to be kept clean and manageable. A cluttered or fragile hardware setup would make demos stressful and unreliable, so we tried to make the system easy to move, plug in, and run without complicated calibration steps.

Finally, safety and practicality influenced our decisions. All components needed to run at low voltage and low power so we wouldn't need special equipment, certifications, or protective gear. By sticking with USB-powered devices and low-power antennas, our system stayed safe, simple, and compliant with the typical rules for student projects on campus.

Overall, these non-technical constraints shaped the project just as much as the technical ones. By keeping the cost reasonable, working with available components, simplifying the design, and making sure the hardware was easy to set up and demonstrate, we were able to build a functional and reliable SDR diversity system that fit within the limits of our resources and timeline.

Time Constraints

Time is one of the most significant and challenging non-technical limitations that we face as an academic engineering team. Unlike industry projects, which may span several months or years and allow for continuous revisions and support, our project must be completed within the rigid

time frame of a university semester. This includes multiple deliverables such as the project proposal, midterm report, hardware and software demonstrations, final report, and end-of-semester presentation. These deadlines are fixed and cannot be moved, regardless of unforeseen issues or development setbacks.

In a real-world scenario, an engineering team encountering delays due to shipping, debugging, or component failure might be able to push back delivery or request more funding. In contrast, we must deliver results by a specific date or risk losing points, failing benchmarks, or missing the opportunity to field test entirely. This has a direct effect on our decision-making process. We must prioritize functionality over perfection, minimize iteration cycles, and ensure that each phase of the project leaves us enough time to test and validate the system before the next deadline. There is no time buffer for trial-and-error once implementation begins.

Furthermore, time constraints limit the scope of what we can realistically accomplish within a single semester. While we would ideally like to build a robust system that includes advanced features such as machine learning-based signal classification or predictive tracking algorithms, we must instead focus on the essential elements that will make the system work reliably. Time limitations also affect our ability to collaborate with external experts, seek out advanced training, or conduct more detailed simulations. With multiple team members balancing other classes, exams, and personal responsibilities, scheduling work sessions can also become a challenge, further compressing the hours we can dedicate to this project each week.

Environmental Constraints

Environmental constraints are another major factor in the design of our passive aircraft detection system. Unlike projects that are designed for indoor use or controlled environments, our system is intended to operate in harsh, unpredictable, and isolated outdoor conditions. This includes exposure to extreme heat during the day, cold temperatures at night, high humidity, rain, strong winds, and the presence of dust, insects, and potentially even wildlife interference. All of these factors can degrade system performance, damage hardware, or cause complete failure if not accounted for in the design.

As a result, every component in our system must be selected and configured with durability in mind. This includes using sealed enclosures that meet IP65 or higher water- and dust-resistance standards, corrosion-resistant connectors, UV-stabilized materials, and temperature-tolerant electronics. Our solar panels must be rugged enough to withstand weather and impact from debris, while our batteries must be housed in compartments that protect against water ingress and overheating. Even small details—such as how cables are routed, or where ventilation openings are placed—can make a difference in whether the system survives real-world deployment.

In addition to hardware challenges, environmental constraints influence system performance from a signal processing and communication standpoint. For example, heavy rain or dense foliage can affect signal reception, while uneven terrain can introduce delays or distortions in data transmission. Solar energy generation can be inconsistent in cloudy or shaded areas, which means that our system must include power management strategies like duty cycling, deep sleep modes, and energy-efficient processing.

Another important consideration is that environmental testing is difficult to simulate accurately in a university lab setting. While we can test for water resistance using spray bottles or run temperature simulations using software, these are not always good substitutes for real-world field testing. Our limited ability to validate our system under the actual environmental conditions it will face makes it even more important that we overengineer for resilience.

Budget Constraints

Budget limitations are an unavoidable reality of student-led engineering projects. Although our sponsor has generously provided funding for parts and tools, the amount is finite and must cover all expenses from start to finish. Every dollar spent on one part of the system means one less dollar available for testing, backup components, or optional features. This creates a continuous trade-off between cost and performance that influences every major design decision.

For example, while we may have considered higher-end microcontrollers or RF front-end modules with better sensitivity and faster processing, their cost would have exceeded what we can afford. Instead, we rely on cost-effective alternatives, even if they offer fewer features or lower performance. The same goes for our communication system: while cellular or mesh networks might offer a viable alternative to satellite communication, they come with recurring data costs or additional infrastructure that would stretch our budget. Starlink SATCOM, while not cheap, offers a single upfront cost and global coverage, which aligns with both our technical and financial constraints.

Budget constraints also affect our ability to iterate and improve. Ideally, we would produce multiple versions of each PCB to test different layouts or shielding strategies. In reality, each fabrication run adds cost and delays, so we are forced to get it right the first time. This often leads to conservative design choices, where we prioritize reliability and compatibility over experimentation.

Another challenge is accounting for hidden or unexpected costs. Shipping fees, tax, broken components, tools, and test equipment all consume portions of the budget that aren't always visible in the planning phase. For this reason, we must maintain a running ledger of all expenditures and include a financial buffer in case of emergency. These practices ensure that we stay on track without compromising critical functionality.

Manpower and Expertise Constraints

Unlike professional engineering firms that include specialized staff for every major discipline, our student team must handle all aspects of the project with limited personnel and varying skill levels. While this offers a great opportunity for learning and teamwork, it also creates bottlenecks when we encounter topics or technologies that no one on the team has worked with before. For instance, not all team members are experienced in embedded programming, signal processing, or RF system design, so we often need to learn these skills while simultaneously trying to implement them.

This constraint has a direct impact on our development timeline and the depth of the system features we can implement. We must allocate time for training, experimentation, and knowledge sharing among team members, which further compresses our already limited schedule.

Moreover, with each member juggling multiple responsibilities—both academic and personal—the amount of time that can be dedicated to the project each week is limited. We cannot assume 40 hours per week of availability like in a job setting.

Another issue is project management. Because we are all relatively new to large-scale technical projects, we are also learning how to communicate effectively, divide tasks, manage documentation, and track progress. Without a dedicated project manager, these responsibilities must be shared, which can sometimes lead to confusion or misalignment. To mitigate this, we hold regular meetings, document our work consistently, and use collaboration tools like GitHub and shared drives to keep everyone on the same page.

Deployment and Accessibility Constraints

Finally, deployment and accessibility constraints are especially important in our project due to the remote and isolated nature of our target environments. The detection units must be easy to transport, assemble, and activate with minimal tools or technical knowledge. Once deployed, they may not be reachable for weeks or even months, so the system must be designed to operate autonomously with self-diagnostics and fail-safes built in.

This imposes strict constraints on hardware integration and software reliability. We must ensure that power systems can function continuously, that enclosures can withstand field conditions, and that communication links remain stable. There must also be a system in place for remote monitoring and updates, allowing us to push configuration changes or receive alerts if a unit malfunctions.

Accessibility also refers to the ease with which the system can be operated by non-technical personnel. While we are building the units ourselves, future users may not have engineering backgrounds, so the deployment process must be intuitive and well-documented. Simple LED indicators, labeled ports, and plug-and-play connections can reduce the risk of setup errors. Similarly, the dashboard interface used to monitor the units should be designed with usability in mind, minimizing clutter and presenting data in a clear, meaningful way.

Chapter 5. Comparison of ChatGPT with Similar Platforms

As artificial intelligence becomes increasingly embedded in academic and engineering workflows, platforms such as ChatGPT, Microsoft Copilot, Google AI, and DeepL have become valuable tools for research, design assistance, and documentation. These systems provide rapid access to technical explanations, debugging support, literature-style summaries, and code generation—capabilities that significantly enhance productivity for engineering students. In the context of Senior Design, where teams must balance hardware development, software implementation, testing, and report writing, AI-assisted platforms help bridge knowledge gaps, clarify complex concepts, and streamline the creation of structured, technically accurate documents. However, while these platforms offer clear benefits, they differ in depth, correctness, contextual awareness, and their ability to aid in engineering decision-making. This chapter compares ChatGPT with Microsoft Copilot and Google AI, evaluating their strengths, limitations, and overall impact on our learning experience during the development of our Radio Diversity System.

5.1 Case study 1: ChatGPT vs Microsoft Copilot

Prompt: “What are good aspects when using Raspberry pi model 4?”

ChatGPT’s response demonstrated a strong technical understanding of the Raspberry Pi 4, offering a comprehensive evaluation of its CPU performance, memory capacity, connectivity options, operating system support, and suitability for real-time signal processing tasks. Importantly, its explanation was context-aware and directly tied to our project goals—such as running multiple SDR receivers, handling continuous audio decoding, and supporting parallel networking operations. The response also highlighted advanced implementation strategies such as using Docker containers, real-time Linux kernels, and optimized Python libraries, which are highly relevant to embedded signal-processing pipelines like ours. The writing quality was clear, concise, and suitable for direct inclusion in a formal engineering document.

However, ChatGPT’s initial response did not explicitly address some important limitations of the Raspberry Pi 4. These include higher power consumption under continuous load, potential thermal throttling without proper cooling, USB bus contention when using multiple peripherals, and limited real-time performance unless kernel modifications are applied. It also did not compare the Pi 4 to alternative platforms such as the NVIDIA Jetson Nano or ESP32-based boards, which could influence design decisions depending on project constraints. Including these drawbacks would strengthen the balance of the evaluation.

In comparison, Microsoft Copilot generated a response that was beginner-friendly and well-structured but less detailed. Copilot effectively summarized the key advantages of the Raspberry Pi 4—such as affordability, broad community support, and GPIO functionality—but lacked the deeper technical analysis required for engineering justification. It did not connect the Pi’s features to our system requirements, such as continuous audio streaming, multiple TCP server operations, or SNR telemetry processing. As a result, Copilot’s output is useful as a high-level reference or quick overview but insufficient on its own for Senior Design documentation.

Conclusion:

Both platforms offer value but serve different roles. ChatGPT excels in producing in-depth, engineering-focused analysis that can be used directly for design justification, literature-style explanations, and complex debugging. Its responses provide actionable recommendations and are tailored to the project’s technical context. Microsoft Copilot, while helpful for quick summaries or simple clarifications, lacks the depth required for academic-level documentation and system-level decision-making. Therefore, for highly technical components of our Radio Diversity System—such as SDR signal pipelines, buffering analysis, diversity algorithms, and system integration—ChatGPT proved significantly more effective as a development and documentation tool.

5.2 Case study 2: ChatGPT vs Google Gemini

Note: (Both AIs were prepped and informed of the components in our system and the background of the project before the prompt was made)

Prompt: “Tell me about the voltage and current requirements of our system, and how to connect our power supply unit to fill those needs”

The response from ChatGPT is very broad and informative, giving good insight into specs of each product and general recommendations. However, it is not always 100% accurate, and can require some background knowledge to verify that what it’s telling you is exact. For example, it cites a 12V connection and a fuse for connecting our Spectrum Analyzer, but we can quickly verify from the SignalHound website that the power input on the BB60D is a USB3.0 power draw straight from the Raspberry Pi (or whichever computer to which it is connected). Thus, we can see that it is not always a perfectly accurate source, but can be super helpful when researching to help get us off the ground. Best practice is likely to take the info we receive and then go and verify the facts from that.

When asking Google’s Gemini AI, it is a much more vague response, directing me to datasheets and specifications of each component. One consideration it did take is the relation of output watts and the systems Ah value for a 24 hour runtime, but overall it gives me much less info and details to work with.

ChatGPT excelled in technical reasoning and system-level understanding, helping me integrate parts like the Renogy solar components into a cohesive design. It remembered the context of my setup, allowing for deeper, more tailored insights—especially when analyzing trade-offs or interpreting specs from datasheets. The conversational format made it easy to refine ideas and ask follow-up questions without starting from scratch.

Google Gemini, on the other hand, was more useful for real-time information, including datasheet access, availability, and pricing. It was especially helpful when checking for newer model releases. However, it lacked the depth of system-level understanding that ChatGPT provided, and often required me to restate details about the project. Overall, ChatGPT was better suited for design-oriented tasks and technical evaluation, while Gemini worked best as a search-driven tool for quick data gathering. Using both in tandem allowed me to cover both breadth and depth in my component research. Thus, despite ChatGPTs occasional misprints, I find it is a much better starting point than what Google Gemini gave.

5.3 Case study 3: ChatGPT vs Grok

For this part of the project, I used both ChatGPT and Grok AI to compare how well each one could help me with a technical question we needed to answer for our aircraft detection system. I asked them both for antenna recommendations and how those antennas would work with our setup. Right away, I noticed some big differences in the way the two AIs responded—not just in the information they gave, but in how easy it was to understand and use their answers. ChatGPT gave me a very clear, well-organized explanation. It recommended using a Log-Periodic Dipole Array (LPDA) antenna and explained why that type was a good match for our project. It talked about the frequency range, signal direction, durability, and even gave me a simple step-by-step explanation for how the antenna could be set up. The tone was straightforward, easy to follow, and made it super clear how the information could be applied to our real-world system. As someone who's still learning a lot of this stuff, I really appreciated how user-friendly it was. I didn't have to spend extra time researching terms or trying to figure out what the response meant—it just worked for what I needed.

Grok's response was definitely more technical and in-depth, which at first seemed impressive, but after reading it a few times, I realized it was a little overwhelming. It went deep into concepts like TDOA (time difference of arrival), signal triangulation, and placement logic. While this kind of detail might be useful for someone with more experience in RF systems or advanced signal processing, I found it harder to pull out the key points I needed. Grok also didn't give me a clear antenna recommendation right away—instead, it focused more on the general approach and theory behind setting up an antenna network. That's valuable information, but it didn't help me move forward as quickly. I felt like I'd have to summarize or rewrite Grok's response just to use it in our report, whereas ChatGPT's answer was basically copy-and-paste ready after a bit of editing.

Overall, I think both AIs have their strengths. Grok is good for detailed, technical reasoning and could be really useful for backing up a decision or digging deeper into a topic. But when it comes to speed, clarity, and practical use, ChatGPT worked better for me. It felt more like a helpful teammate that understood the level I was at and gave me the information in a way that I could immediately use. For an undergraduate project like ours, where we're trying to balance learning and execution, that made a big difference. I'd definitely keep using ChatGPT as my main tool and maybe use Grok on the side for cross-referencing or if I ever need to dive deeper into something really technical later on.

Chapter 6. Hardware Design

6.1 Power Supply System

The system is designed for full autonomy using the EcoFlow RIVER 2 Solar Generator (PV110W), an integrated solar power solution ideal for continuous operation in remote areas. This all-in-one system combines a 256Wh LiFePO4 (LFP) battery, known for its long lifespan and safety, with a 110W portable solar panel. The solar panel efficiently charges the power station, achieving a full recharge in approximately 2.9 hours under optimal conditions. The RIVER 2 features multiple output ports, including AC, DC, and USB, providing regulated power to subsystems such as the Raspberry Pi, Signal Hound BB60D, and other electronics. Its Battery Management System (BMS) offers protection against over-voltage, overloading, over-temperature, short circuits, low-temperature, low voltage, and overcurrent, ensuring the safety of connected devices. This compact and lightweight solution ensures reliable operation under various environmental conditions, making it suitable for long-term deployment in off-grid locations.

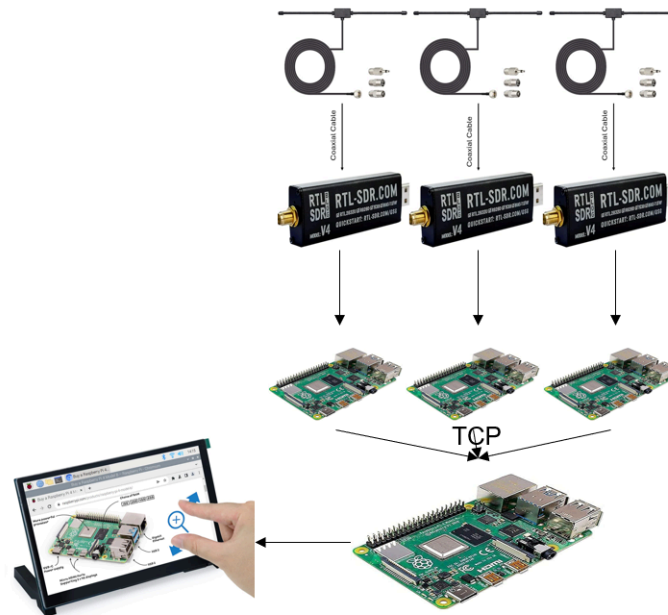


Figure 6.1 Power Subsystem Wiring Diagram

Each component should be properly grounded to prevent electrical noise and ensure stable operation.

- A USB-A connection will power the Raspberry Pi.
- A USB-C power adapter (5V 3A) is needed for the Samsung S24, with proper circuit protection.
- A USB3.0 direct connection from the Raspberry Pi is used for the Signal Hound BB60D.

6.2 Broadcast & Receiver RF Subsystem

6.2.1 Broadcast Hardware

To evaluate diversity techniques, the system requires a consistent RF broadcast source. Two approaches are being used. First, existing over-the-air signals such as commercial FM radio (88–108 MHz) and ADS-B at 1090 MHz provide free, always-on carriers that the system can capture without the need for extra transmitters. These are ideal for outdoor testing since they cover large areas and are easily detectable with even low-cost antennas.

For controlled indoor testing, however, a dedicated broadcaster is needed. The team can use a small USB-based SDR such as the HackRF One or LimeSDR Mini in transmit mode, configured to output a low-power FM carrier or test tone into a shielded enclosure. This allows the team to generate repeatable signals at chosen frequencies without interfering with public spectrum. The SDR-based transmitter connects via USB to a laptop, which runs GNU Radio or SDRangel to define the waveform. This approach avoids the cost of a professional RF generator while still offering flexibility for custom scenarios.

6.2.2 Antenna Schematic (Active/Passive)

Signal collection is handled by three antennas, each mounted on adjustable tripods. Their wideband coverage (800 MHz to 6 GHz) makes them compatible with FM, ADS-B, and custom SDR broadcasts. The antennas are oriented at slightly different angles to enable spatial diversity, with one facing the source directly and the others offset. Each antenna connects to its SDR via high-quality SMA-terminated coaxial cables, kept equal in length to minimize timing mismatches. In the test setup, antennas are placed several feet apart to accentuate spatial differences, but they can also be clustered side-by-side for direct comparisons of diversity processing, like frequency or time diversity. For indoor work, they can be positioned in the same RF chamber as the broadcaster, ensuring clear reception without external interference.



Figure 6.2 Passive Antenna Setup

6.2.2 Spectrum Analyzer Connections

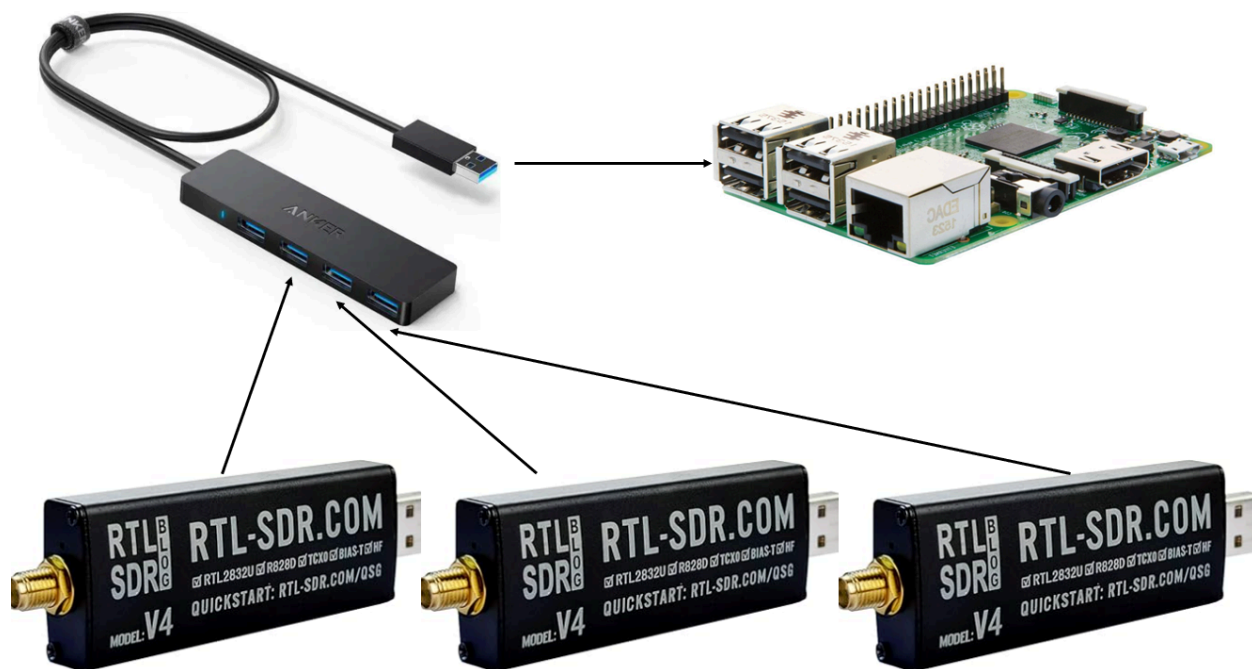


Figure 6.2.2 Spectrum Analyzer

The receiving side uses three RTL-SDR v3 dongles. These plug directly into a powered USB 3.0 hub, which is then connected to the Raspberry Pi 4. The powered hub is essential because the Pi cannot reliably supply three SDRs with sufficient current. Each SDR is locked to its respective antenna feed, and the Pi runs software such as RTLPower or custom Python/GNU Radio scripts to capture and analyze incoming signals.

Drivers are pre-installed on the Pi, and the devices are configured with unique serial numbers so the software can differentiate between them. This ensures that each SDR consistently corresponds to a specific antenna, avoiding ambiguity during multi-input experiments.

6.3 Data Processing and Output System

Once the signals are received by the SDRs, they are processed by the Pi, and sent to the Touchscreen for UI and output.

6.3.1 Raspberry Pi

The computational core is a Raspberry Pi 4 Model B with 4 GB RAM. It provides enough processing capacity to manage three simultaneous SDR data streams while also running the diversity algorithms and updating the user interface. The Pi is housed in a ventilated enclosure with passive heatsinks to prevent overheating during extended captures.

The three SDRs connect to the Pi via the powered USB hub, and the Pi itself is connected to the 7-inch touchscreen display via the DSI ribbon cable. This direct interface avoids the latency and power issues associated with HDMI-based touchscreens. The Pi runs Raspberry Pi OS with custom Python software to handle both SDR processing and the user interface.

6.3.2 Touchscreen User Interface

A Raspberry Pi 7-inch official touchscreen serves as the operator display. It provides real-time visualization of the three incoming signals, showing relative strength on simple bar graphs or line plots. The UI includes 3 selectable modes: threshold mode, automatic mode, and manual override. In manual mode, the user can tap one of the three inputs to force the system to display or analyze only that signal.

The UI is developed in Python using PyQt or Tkinter for the graphical layout. Signal strength values are updated continuously from the SDR input, and diversity mode selection changes how the backend software processes and presents data. Touch events from the screen are mapped directly into software callbacks, making mode switching intuitive for operators.

Chapter 7. Software Design

7.1 Use Case Diagram

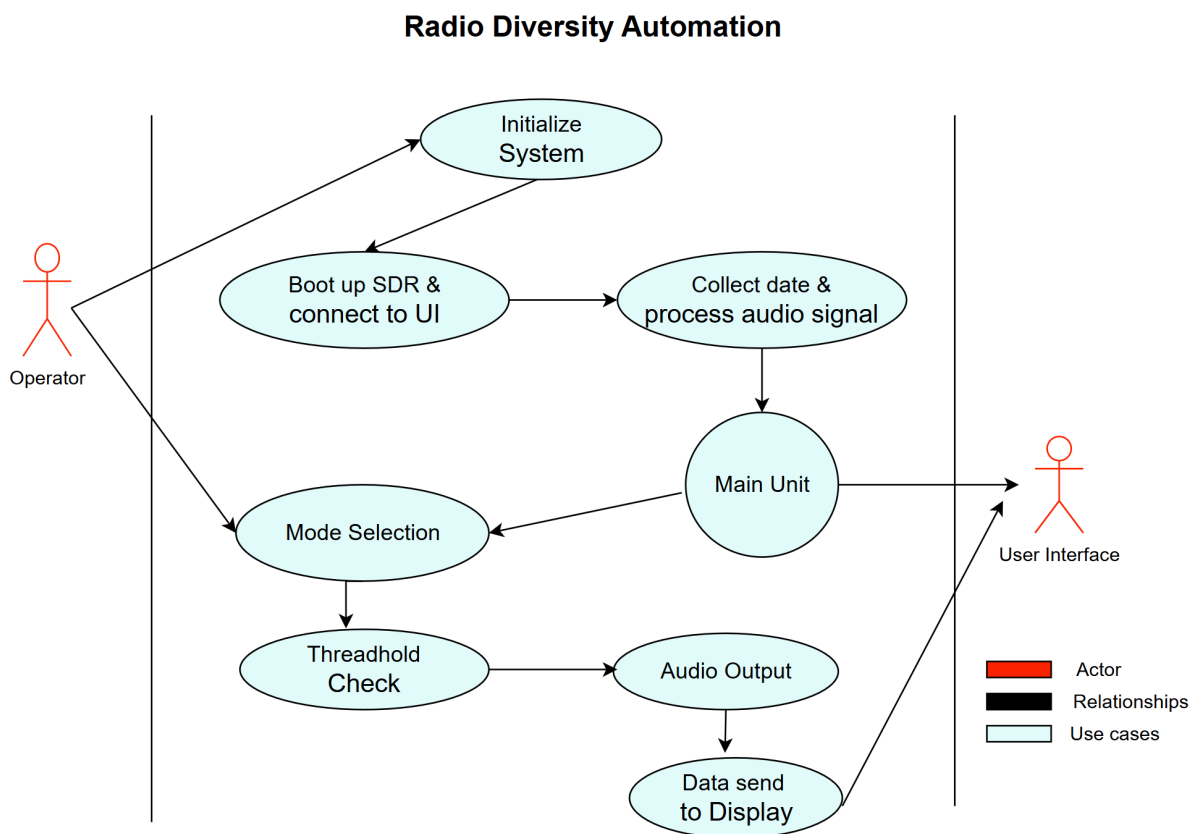
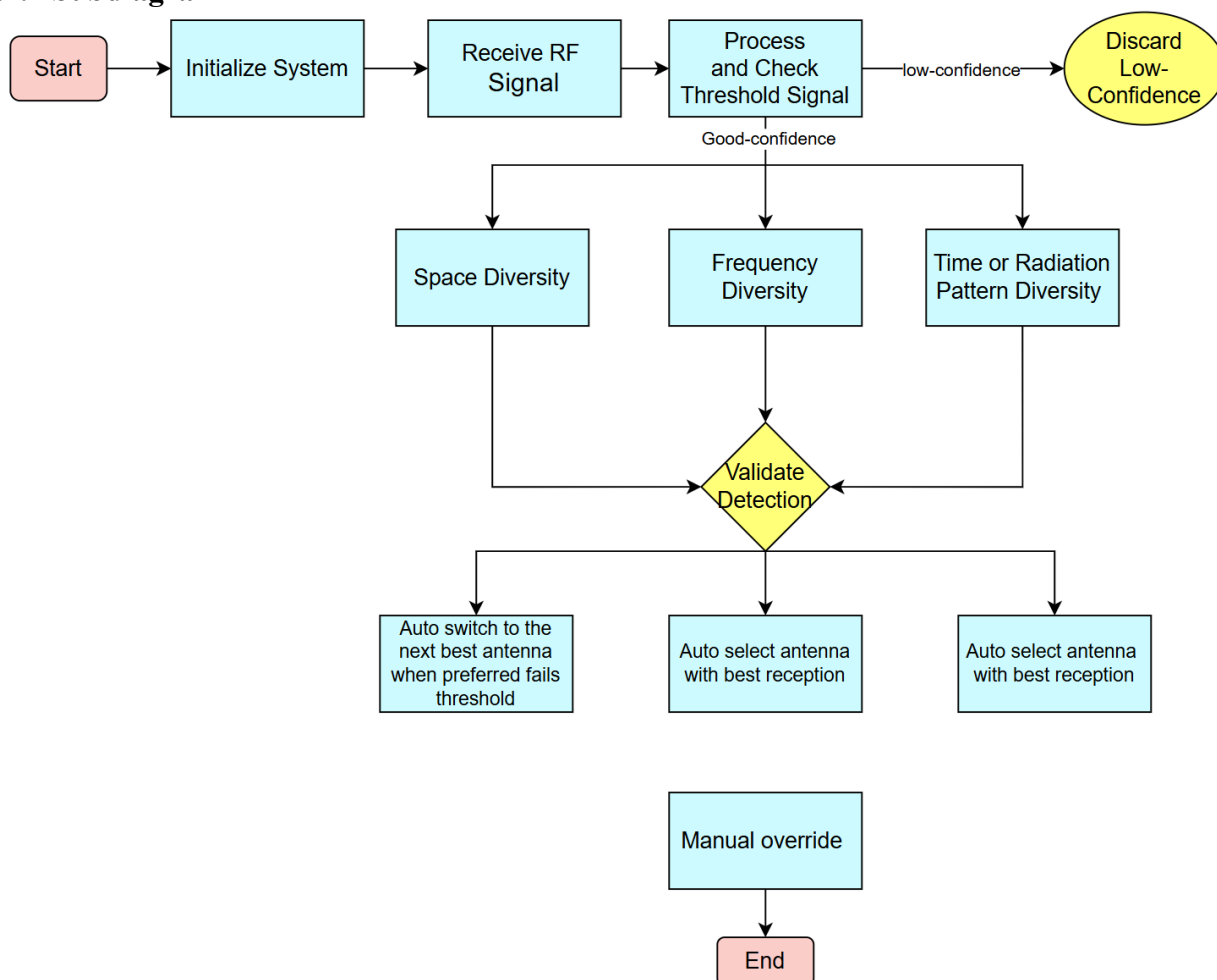


Figure 7.1 Software Use Case Diagram by Authors

The use case diagram illustrates the interaction between the Operator, User Interface, and the system components. The Operator initializes the system, boots the SDR receivers, and selects the operating mode. Each SDR collects RF data and processes the audio signal before sending it to the Main Unit. Based on the selected mode or threshold conditions, the Main Unit determines which SDR provides the strongest and most reliable signal, then outputs the audio accordingly. The User Interface continuously displays system status and signal metrics, providing real-time visibility and control for the user.

7.1.2 Subdiagram

**Figure 7.1.2** Subdiagram

This sub-diagram outlines the internal decision-making flow used in the system. After initialization, the system receives RF signals and evaluates their confidence level. Signals that pass threshold checks are processed using the chosen diversity technique—Space, Frequency, or Time Diversity. The system then validates the detection based on signal quality metrics. If the selected antenna fails to meet the required threshold, the system automatically switches to the next-best option. When a strong signal is confirmed, the system selects the antenna providing the

best reception. Additionally, a manual override option allows the operator to bypass automation and directly control antenna selection.

7.2 State Diagram

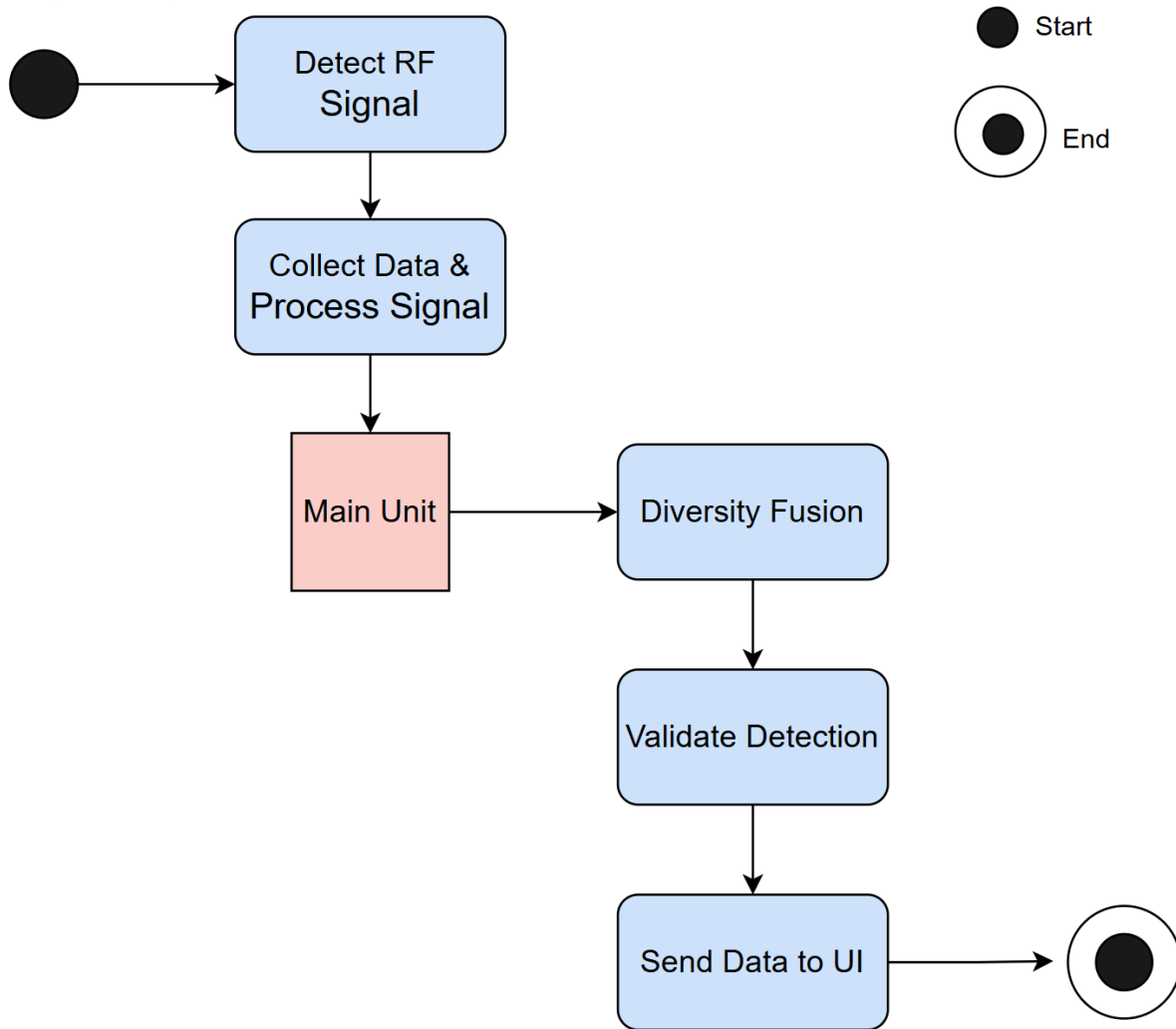


Figure 7.2 Software State Diagrams by Authors

This stage diagram illustrates the full data path in the wireless SDR system. The Receiver Pi initializes its RTL-SDR, configures reception settings, and continuously streams IQ-derived audio frames and telemetry metrics to the central backend via TCP. The central unit runs parallel threads to accept audio and telemetry, buffer incoming frames, perform diversity logic, and route the selected audio for playback. Meanwhile, the UI connects to the backend, displays live signal metrics, and sends user commands back to the system. This flow ensures real-time synchronized communication across all components.

7.3 ERD

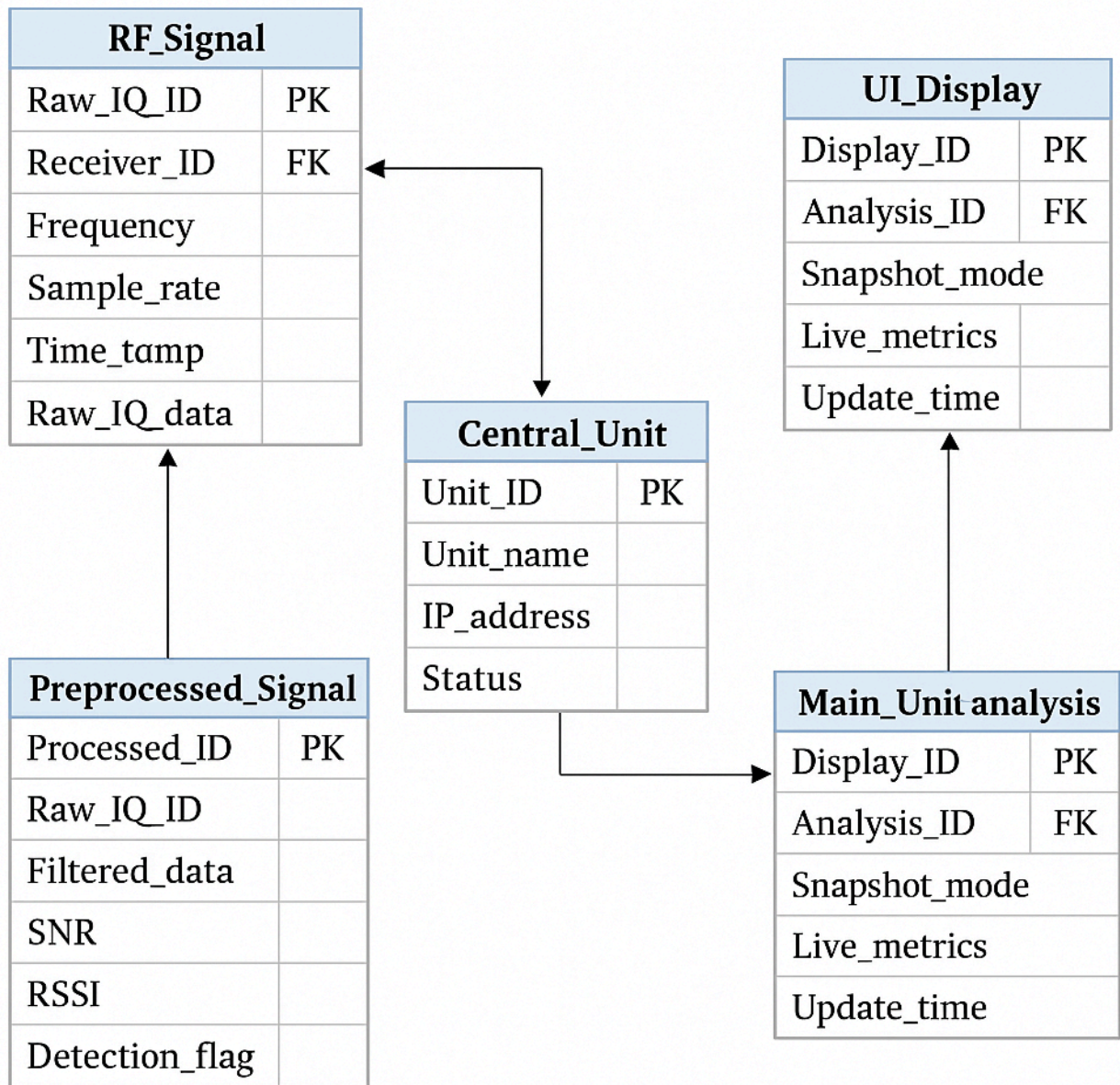


Figure 7.3 ERD

The ERD models how signal and system information flows through the radio diversity architecture. Raw RF samples captured by each receiver are logged in the RF_Signal table and linked to processed signal metrics in Preprocessed_Signal. The Central_Unit manages the identity, IP, and status of the processing node. Diversity analysis results are stored in Main_Unit_Analysis, which is then connected to UI_Display to provide real-time visualization of live metrics and snapshots. This structure ensures traceability from raw IQ acquisition to final UI output.

7.4. Activity Diagram

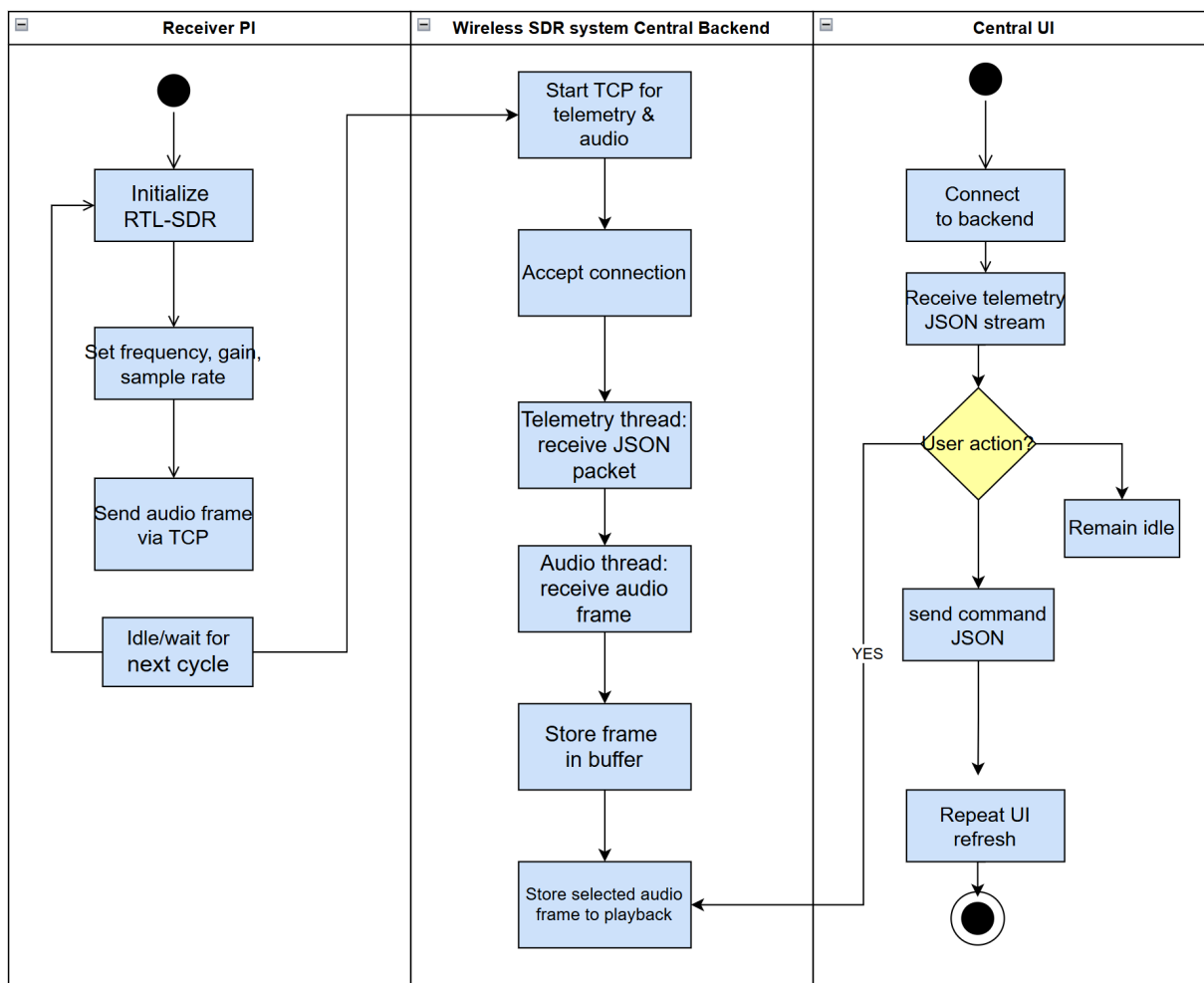


Figure 7.4 State Diagram

The activity diagram outlines the main signal-processing workflow of the diversity automation system. After detecting RF energy, each unit gathers IQ data and extracts key metrics before sending them to the main unit. The main unit conducts diversity fusion to determine the most reliable signal path, validates the detection quality, and determines whether switching is needed. The validated result is then delivered to the UI, providing real-time insight into the receiver performance and system actions.

7.5. User Interface (UI) Design

The User Interface (UI) is an important part of our SDR Diversity System because it is the main way the operator sees what is happening inside the system in real time. Since we shifted away from the original aircraft-detection project and changed the scope toward SDR diversity, we redesigned the UI to fit the new goals. Instead of a web dashboard or a cloud-based display, we chose to build a local Python Tkinter interface that runs directly on the Raspberry Pi 4. This decision made the system easier to test, easier to control in person, and much more lightweight

overall. The Raspberry Pi is already doing a lot of work with three SDR inputs, so keeping the UI simple but still useful was one of our priorities.

The updated UI is meant to be easy to understand at a glance, especially when running experiments or moving the transmitter around. It shows the status of all three SDR receivers, basic signal information, and a live spectrum visualization that helps us see how the signal changes over time. Everything is designed around the 5-inch HDMI touchscreen, which allows us to operate the system with just taps instead of needing a keyboard or mouse. This made the entire setup more convenient and portable, especially when we took the Pi out of the lab for testing.

The UI supports the following features during operation:

- Live signal strength readings from all three SDRs
- A clear indicator of which SDR is currently selected
- A real-time FFT plot of the incoming radio signal
- A scrolling text log showing system events
- Simple Start/Stop controls

These features help us not only run the system but also understand what the diversity logic is doing at any moment.

Table 7.1 UI Breakdown

UI Component	Description	Implementation
Dashboard	The main overview screen that shows general system health and the status of all three SDR receivers. It helps the user quickly see which receivers are active, which one the system is currently using, and whether anything requires attention.	Python (Tkinter)
Signal Panels	Individual panels for each SDR that display real-time metrics like RSSI or strength estimates. These panels update constantly while the system is running and visually highlight which SDR the diversity logic selects.	Tkinter labels with dynamic color updates
Spectrum Plot	A live FFT visualization that	Matplotlib embedded inside

	gives the user a clear picture of the RF activity coming into the system. This plot updates at a controlled pace and helps with understanding how the signal changes due to movement or interference.	Tkinter using FigureCanvasTkAgg
Status Log	A scrolling text window that prints real-time system events, warnings, SDR connection changes, and diversity switching decisions. This helps with troubleshooting and understanding how the system behaves over time.	Tkinter Text widget
Control Buttons	Touch-friendly Start, Stop, and Refresh buttons that let the operator control the system without needing a keyboard or mouse. These buttons are kept simple and spaced out for easier use on a small touchscreen.	Tkinter buttons

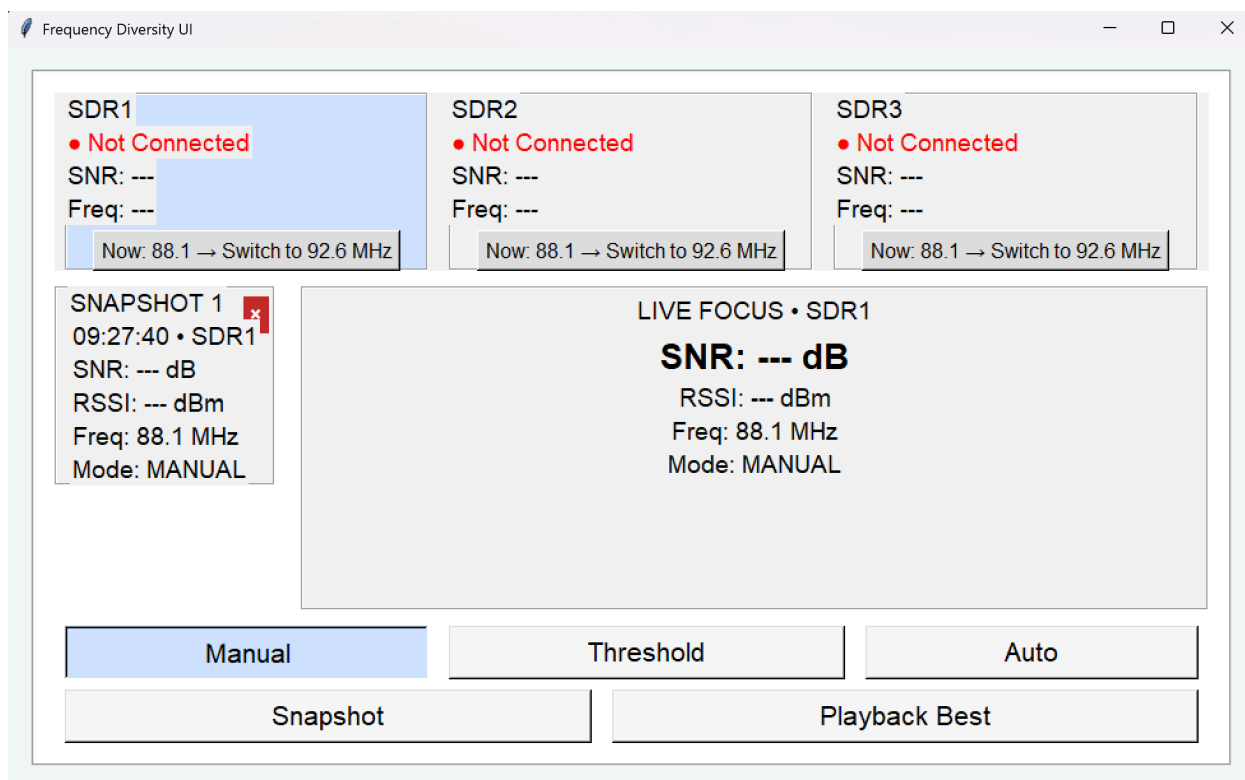


Figure 7.5: UI

Dashboard Functionality

The Dashboard acts as the “home screen” for the entire UI. Each of the three SDRs gets its own panel on the screen, and each panel changes color depending on its status. For example, if an SDR is connected and sending valid data, the panel turns green. If something goes wrong or the SDR stops responding, the panel turns red. This makes it easy to see at a glance which receivers are active, especially during tests where we walk around and vary the transmitter’s position.

The Dashboard also shows extra information like the current diversity mode, any system warnings, and the identity of the SDR that the backend has selected. Even though this information might seem simple, it helped us a lot while troubleshooting, since we could immediately tell if something was wrong before digging deeper into logs.

Signal Panels

The Signal Panels are updated continuously using a background thread. This prevents the UI from freezing, which can happen if long-running tasks run directly in the Tkinter main loop. Each panel displays updated signal metrics every second, including things like RSSI estimates. The selected SDR is highlighted so the user can easily follow how the diversity logic behaves in real time. This was especially helpful when physically moving around the lab or adjusting antennas, because we could immediately see how the signal reacted.

We originally thought about adding more advanced metrics, but since we were using the Raspberry Pi 4 and wanted to keep everything running smoothly, we stuck with the essentials. The simpler design ended up being more stable and easier for anyone in the group to understand.

Spectrum Plot

One of the strongest features of our UI is the real-time spectrum plot. We embedded a Matplotlib graph inside the Tkinter window, which shows the frequency distribution of the incoming signal. The spectrum plot updates at a balanced rate so that the Raspberry Pi does not get overloaded, but it's still fast enough to show subtle changes in the RF environment.

During testing, the spectrum plot helped us see how the signal changed depending on distance, interference, or even small adjustments to the antennas. It made the system feel much more interactive and gave us visual confirmation that the SDRs were receiving the broadcast properly.

Status Log Section

The Status Log records everything that happens behind the scenes. This includes:

- When an SDR connects or disconnects
- When the diversity logic selects a different receiver
- Any warnings from the backend
- General system messages

The log scrolls automatically and gives us a simple way to track behavior during longer tests. We relied on this a lot while debugging because it made issues easier to identify, such as when a receiver temporarily dropped or when networking delays affected the audio output. Even though the log seems basic, it became one of the most important tools for understanding system behavior.

Control Panel

The UI includes three large buttons at the bottom of the screen:

- **Start** – begins the monitoring process and enables UI updates
- **Stop** – safely stops signal monitoring and halts the threads
- **Refresh** – checks all three SDRs and updates the dashboard

Since the UI is used on a touchscreen, we made these buttons bigger and spaced them out to avoid accidental taps. It also just makes the interface look cleaner and easier to use.

Technology Stack and Responsiveness

The entire UI is built using only a small number of Python libraries, mainly:

- **Tkinter** for all visual components
- **Matplotlib** for plotting the spectrum
- **Threading** and **Queues** for background processing

- **Custom Python logic** to communicate with the backend

We chose Tkinter because it's simple, lightweight, and already installed on the Raspberry Pi OS. There's no need for a browser, no extra dependencies, and no heavy frameworks that slow the Pi down. While Tkinter isn't the flashiest GUI toolkit, it perfectly fit our goal of keeping the system easy to run and easy to maintain.

Overall, the UI stays responsive even when all three SDRs are active, which shows that our decision to keep things simple worked out well.

Summary

In the end, the User Interface turned out to be an important part of our project because it ties the whole system together in a way that's easy to use and understand. It shows live signal updates, highlights the selected SDR, logs important events, and provides basic controls. Even though the interface is simple, it fits our project perfectly by giving us the information we need without slowing down the Raspberry Pi. The UI helped us observe the behavior of the diversity algorithm in real time, and it made testing and troubleshooting a lot easier for our team.

Chapter 8: Prototyping and Testing

8.1 Hardware Testing

Hardware testing for the Radio Diversity System focused on validating the performance, reliability, and consistency of all physical components involved in the distributed SDR receiver network. This included the RTL-SDR Blog receivers, antennas, Raspberry Pi units, power delivery hardware, and the wireless networking environment. Initial tests began with individual component verification, where each RTL-SDR was connected directly to a laptop or a Raspberry Pi to confirm correct device recognition, stable USB communication, and proper RF front-end functionality. Each SDR was tested across multiple frequencies to ensure wideband reception worked as expected, and gain settings were adjusted to determine the optimal configuration for clean demodulated audio under varying signal strengths. Antenna testing was also performed to evaluate reception quality, including comparisons between different antenna placements, orientations, and distances relative to the transmitter.

Raspberry Pi hardware testing focused on ensuring proper boot behavior, CPU stability, and consistent performance under real-time signal processing loads. We monitored CPU utilization, thermal behavior, and power draw while running full demodulation and network streaming tasks. These tests confirmed that the Raspberry Pi 4 had sufficient processing capability to handle continuous FM demodulation and TCP audio transmission without overheating or throttling. Additional testing validated that the Pi could maintain stable USB communication with the RTL-SDR, since insufficient power or poor grounding can cause tuner dropouts or device resets. To ensure robust operation, each Pi was tested with both stock power supplies and higher-rated adapters to eliminate the possibility of undervoltage errors.

Wireless networking reliability was an important part of hardware testing, as two of the three SDR receivers operated over WiFi. Tests were conducted under various network

conditions—including ideal line-of-sight placement, obstructed indoor environments, and congested WiFi channels—to characterize the impact of wireless conditions on packet timing and audio delivery. We measured signal strength, packet loss, and connection stability while streaming audio to the Main Unit. These evaluations helped isolate network-induced jitter from software buffering issues. Wired Ethernet testing was also performed to establish a baseline for ideal performance and to demonstrate the differences between wired and wireless receivers before software-side buffer corrections were implemented.

We also performed whole-system physical testing, running all SDR receivers simultaneously while moving the transmitter around the test environment. These tests validated that the receivers maintained RF lock, delivered continuous data under mobility, and did not exhibit unexpected dropouts. During these experiments, we monitored the physical stability of antenna connectors, USB ports, and power cables to identify any intermittent hardware failures. This was important because loose connectors or poor RF coupling can mimic software issues by causing sudden SNR drops or spurious disconnect events.

Extended-duration hardware stress tests were conducted by running the entire system for several hours to ensure that none of the hardware components overheated, throttled, or became unstable over time. Throughout these runs, we monitored the SDR tuner temperatures, Raspberry Pi thermal performance, and USB bus reliability. The system remained stable for long periods, confirming that the hardware platform is capable of supporting real-time multi-receiver diversity tasks without requiring additional cooling, shielding, or hardware modifications.

Overall, the hardware testing phase demonstrated that the physical SDR receivers, the Raspberry Pi units, and the networking hardware all perform reliably under continuous operation. These tests also ensured that the observed system latencies were not caused by hardware failures, but rather by software buffering behavior, leading us to the software improvements documented in the subsequent sections of the report.

8.2 Software Testing

Extensive software testing was performed on the Radio Diversity System to ensure functional correctness, real-time responsiveness, and reliable operation under various RF and network conditions. Testing began at the unit level, where each individual SDR receiver was evaluated independently for proper RF capture, FM demodulation, and telemetry generation. During these tests, each receiver was connected to a controlled FM transmitter, allowing us to verify that IQ samples were correctly processed, SNR and RSSI values were accurate, and the audio output matched the transmitted signal without distortion or timing errors. Telemetry messages were examined for correct formatting, proper updating rates, and consistent delivery to the Main Unit over TCP.

Network testing focused on evaluating communication reliability between the receivers and the Main Unit across wired and wireless connections. Using both real and simulated network conditions, we measured socket stability, packet timing, and the behavior of the system when subjected to jitter, latency, and temporary packet loss. Testing also involved intentionally disconnecting receivers to observe how quickly the Main Unit detected a timeout and whether stale data was cleared properly. This step was essential for identifying the “ghost audio” issue,

where old audio buffered from a disconnected SDR would still play back later; this was later resolved by flushing buffers immediately on disconnect.

Main Unit testing examined the correctness of the mode selection logic. Manual Mode was validated by ensuring that the selected receiver's audio was always chosen regardless of signal conditions. Threshold Mode required confirming that SDR 1 was selected only when its SNR exceeded the predefined threshold and that the system correctly fell back to the strongest alternative when needed. Auto Mode testing involved dynamically varying signal strength across receivers and verifying that the Main Unit consistently selected the SDR with the highest SNR in real time. During these tests, SNR telemetry was cross-checked with the resulting audio stream to ensure the selection engine behaved as expected.

A significant portion of testing focused on audio buffering and playback, especially after discovering severe latency caused by unbounded queues for non-focused SDRs. After implementing bounded buffers and trimming them on SDR switches, we conducted repeated latency measurements under both wired and wireless connections. These tests confirmed that audio delay remained consistently under ten seconds and that SDR switching produced near-instant transitions to live audio without noticeable artifacts or gaps. End-to-end system testing was then performed with all three SDRs operating simultaneously. By physically moving the transmitter between different regions, we generated shifting SNR values and verified that auto mode maintained continuous, high-quality audio output while seamlessly selecting the optimal receiver.

Additional validation included snapshot testing, where recorded audio segments were saved and replayed to ensure the history buffer and playback functions behaved correctly without interfering with live audio. Finally, long-duration stress testing was performed by running the entire system for several hours to monitor for issues such as memory leaks, queue buildup, socket instability, or audio drift. After the buffering strategy was redesigned, the system maintained stable performance throughout extended operation, confirming its readiness for real-world use.

8.3 Integration Testing

Integration testing focused on validating the complete end-to-end operation of the Radio Diversity System once all software modules, hardware components, and network connections were combined into a single unified architecture. The goal of this phase was to ensure that each subsystem—SDR receivers, telemetry reporting, audio processing, diversity logic, user interface, and network communication—interacted correctly and reliably under real operating conditions. Testing began by bringing up all three SDR receivers simultaneously and verifying that each Pi was able to initialize its RTL-SDR hardware, establish TCP connections to the Main Unit, and stream both audio and telemetry at the required rates. This confirmed that the receivers could operate continuously and concurrently without resource conflicts or socket failures.

A major part of integration testing involved validating the behavior of the system under realistic RF conditions. To achieve this, we moved the transmitter physically around the environment to generate dynamic changes in signal strength across the three receivers. This allowed us to test how well the Main Unit processed multiple incoming streams at once and whether it consistently

made correct diversity decisions based on real-time SNR values. The transitions between receivers—especially in Auto and Threshold modes—were closely monitored to ensure smooth audio switching with minimal perceptible delay. Before the buffer fixes were introduced, these tests exposed significant latency issues when switching between receivers; after implementing bounded buffers and queue trimming, integration tests confirmed that the Main Unit correctly switched to the optimal SDR with near-live audio output.

Network integration was another key aspect of this testing phase. Since the system used a combination of wired and wireless links, we evaluated the full system under mixed network conditions to determine how well telemetry, audio, and control signals propagated through the network simultaneously. This included observing how the Main Unit handled packet timing differences between wired and wireless receivers, as well as verifying that the disconnect detection and reconnection logic behaved correctly in a multi-receiver environment. Scenarios involving temporary WiFi drops, router interference, and SDR reboot events were tested to ensure the system recovered gracefully without crashing or producing corrupted audio.

The User Interface also underwent integration testing to validate that it accurately reflected real-time system status. This included confirming that SNR values, RSSI readings, focused SDR indicators, and mode settings remained synchronized with the Main Unit's state. We also tested the UI's ability to send commands—such as frequency changes, mode changes, and snapshot requests—and verified that the Main Unit correctly forwarded these commands to the appropriate SDR and produced immediate, expected behavior. This step confirmed that the UI and backend communicated reliably under live, multi-receiver conditions.

Finally, extended integration tests were run with the entire system operating continuously for several hours. These long-duration tests ensured that no subsystem introduced memory leaks, socket failures, timing drift, or cumulative latency. After implementing the corrected buffering strategy, the system demonstrated stable, synchronized behavior across all receivers with consistent audio output and accurate UI reporting. Overall, integration testing validated that the combined hardware and software architecture functioned as a cohesive, real-time radio diversity system capable of maintaining stable operation in realistic conditions.

8.4 Enclosure and Environmental Housing

The enclosure and environmental housing for our Radio Diversity System were designed to protect the SDR receivers and Raspberry Pi units while ensuring reliable performance in real-world operating conditions. The hardware enclosure plays a critical role in maintaining system stability by providing physical protection, preventing accidental cable disconnections, and shielding sensitive components from environmental exposure. During development, we evaluated several enclosure materials and configurations to determine the optimal balance between durability, ventilation, ease of access, and RF performance. The final design consisted of a compact, lightweight housing capable of securely holding the RTL-SDR Blog receivers, antennas, and Raspberry Pi boards with proper cable routing to prevent strain on USB and RF connectors.

Thermal management was a key consideration in the enclosure design. Since the Raspberry Pi and SDR hardware generate heat during extended operation—particularly during continuous

demodulation and network streaming—we tested different ventilation patterns and airflow configurations. The housing incorporates multiple ventilation openings to allow passive cooling, preventing thermal throttling and ensuring stable, long-duration performance. During environmental testing, temperature levels inside the enclosure were monitored to confirm that the system remained within safe operating limits even during multi-hour operation. These tests demonstrated that passive airflow was sufficient and that no additional fans or external cooling solutions were required.

Environmental robustness was also evaluated through a series of tests simulating typical deployment conditions. The enclosure was tested for vibration resistance, handling stress, and stability when mounted in various positions. During mobility tests—such as moving the transmitter around and relocating the SDR receivers—the enclosure maintained secure component placement without introducing mechanical noise or interrupting SDR functionality. We also assessed the system’s performance under varying indoor environmental conditions, such as changes in ambient temperature, humidity, and placement near electronic interference sources. The housing successfully protected the electronics from dust, accidental contact, and minor environmental fluctuations, demonstrating its suitability for controlled indoor deployments.

Additionally, the enclosure design considered RF performance, particularly antenna placement and cable routing. To avoid signal degradation, antennas were positioned externally with sufficient spacing to minimize coupling and ensure consistent signal reception. Cable strain relief and port accessibility were included in the design to facilitate easy maintenance and prevent damage to the SDR receiver connectors. These considerations were verified through repeated RF and connectivity tests, ensuring that the enclosure did not negatively affect SNR, RSSI, or overall system sensitivity.

Overall, the enclosure and environmental housing testing confirmed that the system’s physical design supports stable, reliable operation while protecting the hardware from mechanical and environmental stressors. The final enclosure provides a balance of durability, thermal efficiency, RF transparency, and ease of service, making it well-suited for extended operation in a variety of indoor test environments.

Chapter 9: Test Plan

This chapter explains the testing approach we used to verify that our SDR Diversity System works the way we intended. Since our project is no longer based on aircraft detection, BB60D analyzers, or SATCOM logging, the entire test plan had to be redesigned to match our updated architecture. The main purpose of testing now is to make sure that all three SDR receiver units, the central Raspberry Pi, the audio streaming pipeline, the diversity logic, and the UI behave correctly under realistic conditions.

Our testing goal was pretty simple: prove that the system can reliably pick the best signal out of three receivers in real time while streaming audio and updating the UI without freezing. To get there, we tested each part separately and then ran full-system tests where all receivers streamed data at the same time..

9.1 Raspberry Pi as the Target Hardware Platform

The Raspberry Pi 4 remains our main hardware platform, but the way we use it in this project is very different compared to the original aircraft-detection design. Instead of running heavy RF processing tasks or SATCOM scripts, the Pi now acts as a lightweight receiver controller that handles RTL-SDR communication, FM demodulation, telemetry extraction (RSSI/SNR), and streaming audio to the main unit.

During testing, we focused on making sure each Raspberry Pi could:

- Detect and initialize its RTL-SDR
- Tune to the correct broadcast frequency
- Demodulate FM audio smoothly
- Extract SNR/RSSI telemetry once per second
- Stream audio and telemetry to the central Pi over Wi-Fi or Ethernet
- Run for long periods without overheating or crashing

To verify this, each Pi was tested individually at first. We monitored CPU usage, checked for thermal throttling, looked for RTL-SDR disconnects, and confirmed that the Pi could run for extended periods without dropping packets. These tests helped us confirm that the Raspberry Pi 4 is capable of handling its role reliably before connecting all three receivers in the full system.

9.2 Main Unit Testing (Central Raspberry Pi)

Once the receiver units were working, we moved on to testing the central Raspberry Pi, which ends up doing most of the heavy lifting. The main unit accepts three incoming audio streams, reads telemetry from all receivers, and then applies our diversity logic to decide which stream is the best one to play.

The tests for the central Pi included:

- Verifying that all three receivers could connect at the same time
- Checking that audio from each receiver arrived with consistent timing
- Making sure the diversity switching didn't cause audio glitches
- Testing communication between the backend and the UI
- Ensuring that UI updates (SNR, selected SDR, mode, etc.) didn't slow down the system

We spent a lot of time trying different combinations of Wi-Fi and Ethernet to see how latency affected the streams. Wireless receivers occasionally arrived with jitter or delayed frames, so we tested buffer sizes and switching thresholds until the system felt smooth and responsive.

9.3 Diversity Logic and Selection Testing

A big part of the project was making sure the diversity logic actually works in real-time conditions, not just on paper. To test this, we moved the FM transmitter to different parts of the room, placed obstacles in front of the antennas, and intentionally created weak spot zones.

We tracked how the system responded by watching:

- SNR changes on each SDR
- How fast the backend reacted
- Whether the system switched to the correct SDR
- How the audio sounded during switching
- Whether the UI updated without lag

We repeated these tests many times, using both quick signal drops and slow fading changes. This helped us tune the threshold logic in manual mode and validate that automatic mode consistently picked the receiver with the best signal.

9.4 UI and Responsiveness Testing

Because the UI runs on a touchscreen Raspberry Pi and updates in real time, it was important to make sure the interface stayed smooth and didn't freeze when all three receivers were active. For this part of the testing, we focused on:

- Stability of Tkinter during long running sessions
- Refresh rate of UI labels and metrics
- Spectrum plot performance under load
- The timing of backend-to-UI communication
- Touchscreen responsiveness

We ran multiple one-hour and two-hour continuous sessions to see if anything would slow down or break. The UI performed well overall, and the simple layout helped keep CPU usage low.

9.5 Full System Testing

After confirming that each component of the system worked properly on its own, we moved on to full system testing. This is where we combined all three receiver units, the central Raspberry Pi, the streaming pipeline, the diversity logic, and the UI to see how everything performed together in real time. Full system testing was important because it showed us how the system behaved under the same conditions it would face during everyday use. It also helped us identify issues that didn't show up during isolated tests, such as timing mismatches between receivers or small delays caused by Wi-Fi interference.

During these tests, we observed how well all three SDR receivers maintained their connections and whether any of them dropped out under load. We paid close attention to how the system behaved when the signal strength changed, either because we moved the transmitter around the

room or because obstacles temporarily blocked one of the antennas. This let us see whether the diversity logic actually reacted the way we intended. We watched the SNR values rise and fall in real time on the UI and checked if the backend switched to the correct receiver based on those changes. A big part of this process involved listening to the audio playback and checking that switching events didn't cause loud glitches or noticeable interruptions.

We also focused heavily on the performance of the UI during these tests. Since all telemetry and receiver updates were happening at once, we wanted to make sure that the UI stayed responsive and didn't freeze or lag behind. The spectrum plot updated smoothly, and the SNR labels refreshed at the expected rate. Even during long testing sessions where the system ran for more than an hour, the UI continued functioning normally, which gave us confidence that it could handle real-world usage. Full system testing also showed us how the system reacted to connection problems. For example, when we temporarily unplugged or moved one of the receiver Pis, the UI accurately reflected the disconnection and then updated again once the device came back online. Seeing the system recover on its own helped demonstrate that it was stable enough to run without constant manual oversight.

Overall, the full system tests helped validate that all parts of the design worked together the way we planned. It also helped us fine-tune some of the switching thresholds, buffer sizes, and update rates to make the system feel smoother during operation. These tests ended up being one of the most important parts of the project because they showed us that the whole system—not just the individual pieces—could operate reliably.

9.6 Summary

In summary, the revised testing plan for our SDR Diversity System helped us confirm that our overall design actually works in practice. Instead of focusing on aircraft detection or complicated SATCOM integration like the original project, our testing centered around proving that three distributed receivers and a central processing unit could work together as a single system. By running a lot of smaller tests first—such as verifying each Raspberry Pi, checking demodulation stability, and making sure the UI refreshed correctly—we were able to build confidence in individual components before combining them into a full setup.

Once everything was connected, the full-system tests gave us the clearest picture of how the system behaves in real time. We watched how each receiver handled changes in signal strength and how quickly the diversity logic switched to the best available signal. Seeing the UI reflect those updates live, without freezing or slowing down, reassured us that our design choices were practical and realistic for the hardware we were working with. The audio playback sounded smooth during most switching events, and even when we intentionally stressed the network or moved the transmitter into weaker spots, the system adapted and kept running.

These results gave us a better understanding of the limitations and strengths of our design. They also helped us recognize that even though our system is smaller and simpler than the original passive detection concept, it still demonstrates meaningful diversity behavior that could be expanded further in the future. Overall, the testing process showed that our approach is reliable, understandable, and effective for the goals we set, and it gives us a solid foundation as we move into the final stages of the project.

Chapter 10. Administrative Content

10.1 Project Budget and Financing

Our budget for the project is funded by the team and a sponsor. The sponsor will not be sending funds but rather has already picked out parts that we must use and bought them. Through conversation with the sponsor it is unlikely that they will fund anything other than the components they are already buying and thus the team will be handling those costs.

Table 10.1 Budget

Component	Name	Price	Quantity
Spectrum Analyzer	Signal Hound BB60D Real-Time Spectrum Analyzer	\$4835.00	5
Wideband Antenna	Planar Log-Periodic Antenna AX-37A	\$459.95	10
Starlink Terminal	Starlink Mini Terminal	\$599.00	5
Starlink Connection	Roam Unlimited (2 months)	\$330.00	1
Phone	Samsung Galaxy S24	\$799.99	5
Cell Service	T-Mobile Essential Saver Plan (2 months, 5 lines)	\$300.00	1
Laptop PC	HP Laptop 17	\$549.99	1
MCU	Raspberry Pi 5	\$120	5
Total		\$37,804.43	

10.2 Bill of Materials

Attached below is the BOM of all components we have purchased.

Bill of Materials (BOM)				
Item	Description	Quantity	Unit Cost (USD)	Total Cost (USD)
RTL-SDR Blog V4	Software-defined radio receiver	3	\$30	\$90
Dipole Antennas	Omnidirectional antennas for SDRs	3	\$10	\$30
F type to SMA Adapters	Adapters for connecting antennas to SDRs	3	\$3	\$9
7" Raspberry Pi Touchscreen	HDMI+USB touchscreen used for UI	1	\$50	\$50
Walkie Talkie / FM Transmitter	Analog FM broadcast source for tests	2	\$20	\$40
Powered USB Hub	Provides adequate current for 3 SDRs	1	\$20	\$20
Raspberry Pi 4	Central processing unit (Provided by UCF)	3	\$0	\$0
Canakit USB-C Power Supply	Power supply provided at no cost	3	\$0	\$0

Figure 10.1 BOM

10.3 Work Distribution

Table 10.2 Table of Work Distributions

Name	Assembly	Administrative	Signal Processing	User Interface	Hardware System Design
Kim	Secondary - Secondary Team Lead - Assign Roles	Primary - Schedule Meetings - Correspond with Dr. Chan	Secondary - Software Framework - JSON Telemetry Data		
Jurgis				Secondary Design Main UI Layout Visuals	Secondary Secure Screen with Touch

Evan	Primary • Team Lead • Ensure Functional Device		Secondary • Wireless & Audio Implementation • Troubleshooting Issues	Secondary • .wav File Saving and Playback • Freq. Switch for SDRs	Primary • Design & Assembly of hardware • Test all components
------	---	--	---	--	--

10.4 Schedule and Milestones

10.4.1 Senior Design 1

Table 10.3 SD1 Schedule

Senior Design 1	
Week	Description and Dates
1-2	Formulate Ideas and Construct Groups
3	Research/Write Divide and Conquer Document and Form Committee(1/24/25)
4	D&C Meeting with SD Coordinators(1/29/25), Revise D&C(2/7/25), Start 60 Page Report (3/24/25)
5-6	Research/Order Components and Research Software, 15 Page Check-in
7-8	Build/Test Designs, Midterm Optics Showcase(2/25/25), 30 Page Check-In
9-10	More Testing and Optimizing Designs, 45 Page Check-In
11-12	Finish 60 Page Draft(3/24/23), Feedback Meeting, Begin 120 Page Report(4/22/25)
13	Begin Constructing Prototype, 75 Page Check-In
14	Finish Construction of Prototype and Begin Testing, 90 Page Check-In
15	Complete Testing of Prototype, 105 Page Check-In, Final Optics Showcase(4/15/25)
16	Produce Complete Prototype, Finish 120 Page Report(4/22/25)

10.4.2 Senior Design 2

Table 10.4 SD2 Schedule

Senior Design 2

Week	Description and Dates
1	Edit 120 Page Report as Needed
2-3	Start Connecting Devices to a Network of Devices and start on Solar Power Supply
4-5	Research Interconnecting Devices, Continue Making Original Prototype Self-Sustained
6	Finish Converting Prototype to be Self Sufficient
7-8	Test Self Sufficient System, Build Additional Prototypes for Final Report
9-10	Assess and Fix Potential Errors, Finish up Interconnection Software
11-12	Begin Full System Testing, Assess Problems and Begin on Solutions
13-15	Just Keep Testing, Just Keep Testing, Just Keep Testing Testing Testing
16	Final Presentation for the Committee

Chapter 11: Conclusion

As we close out the first phase of our senior design project, it's clear just how far we've come from where we started. What began as a general idea—creating a passive aircraft detection system that could work in remote areas—has grown into a real, well-thought-out design supported by hours of research, testing, team discussions, and problem-solving. Throughout the semester, we've worked to turn this idea into a working system that can actually solve a real-world problem. It hasn't always been easy, but the progress we've made gives us confidence that we're on the right track as we move into Senior Design 2.

The original goal of our project was to design a system that could detect aircraft flying through areas without radar coverage. These areas—especially those along borders or in low-infrastructure regions—don't have the ability to monitor airspace effectively, which creates major security risks. Our system aims to fill that gap by using passive RF detection. That means instead of sending out signals like traditional radar, we just listen for signals that aircraft naturally emit. This approach is not only more discreet, but also significantly cheaper and more energy-efficient. It's a great solution for places that need coverage but don't have the resources for expensive, power-hungry radar systems.

From the start, we were also focused on how to make our system scalable. We knew it wasn't just about building one detection unit that worked. The real challenge was making several of them work together. That meant figuring out how to synchronize them, how to share data between units, and how to build software that could process inputs from multiple points to estimate an aircraft's position. We spent a lot of time researching how to do this using methods like time difference of arrival (TDoA), which lets us figure out where an aircraft is based on when each unit detects its signal. It's exciting because it transforms our system from just detecting something is in the air to knowing where it is and where it's going.

As we dug deeper into the design, we had to make some big decisions about what parts and tools we would use. We ended up selecting the Raspberry Pi 4 as our main processor. It gave us the flexibility and computing power we needed while still being low-cost and energy-efficient—two really important factors for a field-deployed system. We paired that with other critical components, like the Signal Hound BB60D for RF spectrum analysis and a Log-Periodic Dipole Array (LPDA) antenna, which we chose for its ability to pick up a wide range of frequencies with strong directional gain. All of this was powered by a solar-battery setup so that each unit could work in remote locations without needing to be plugged in.

One thing we're especially proud of is how much we were able to test and validate, even without building a custom PCB. Since the sponsor didn't require us to fabricate a board, we worked with modular setups on breadboards. We tested the microphone and amplifier circuits, hooked them up to the Raspberry Pi, and monitored signal strength using both software and simple visual

tools. This allowed us to catch issues early, like signal noise and power instability, and make adjustments before integrating more complex components.

We also learned a lot from building and testing the software. We wrote scripts that allowed us to collect data, process it, filter out noise, and even visualize signals in real time. We tested serial communication between the Pi and other devices and ran stress tests to see how well the system would hold up under continuous operation. These tests helped us think more deeply about how the final system will function in the field, and they gave us the foundation we need for full system integration in the next phase.

Another huge part of this semester's work was figuring out how to protect the system from the elements. Since the units will be deployed outdoors, we needed to design a case that could handle rain, heat, wind, dust, and even wildlife. We spent a lot of time looking at different enclosure materials and design features—like waterproof cable glands, breathable membranes for passive cooling, and shock-mounted components to absorb vibrations. It might seem like a small thing, but the enclosure is what will ultimately keep the system running when no one is around to fix it. With no PCB holding everything together, the enclosure became even more important. It's the physical support system and the first line of defense against weather and wear.

Looking ahead to Senior Design 2, there's still a lot to do, but we're in a good place to keep building. One of our main goals will be to scale the system and test it as a network. That means setting up multiple units, syncing them using GPS, and seeing how well they can work together to detect and locate aircraft in real time. We also plan to build out our base station, where all the detection data will be sent, processed, and displayed on a user-friendly dashboard. This will give us a clear view of how the system is working and whether it's ready for more complex deployment scenarios.

We're also planning to spend time improving system efficiency. We'll keep working on power-saving features, like sleep modes for the Pi, smarter data filtering, and better error handling. We may even look into machine learning for signal classification down the line, depending on how much time we have. There's also room for refining our enclosure with field feedback, stress testing, and potential improvements in layout and heat management.

Beyond the technical side, this project has been an incredible learning experience for all of us. It's challenged us to work as a team, manage our time, and stay organized even when things didn't go according to plan. We've had to learn new tools, figure out how to break down a big idea into smaller, testable steps, and communicate our progress in a way that made sense not just to us but to our sponsor and professors. The fact that this project could have real-world impact made it even more meaningful. Knowing that the system we're building could help secure airspace in underserved regions kept us focused and motivated, even during the tougher parts of the semester.

In the end, this first phase of the project gave us everything we hoped it would. We explored the core concepts, built a working prototype, and proved that the system is viable—not just on paper, but in real-world tests. We identified the key challenges ahead and have a clear roadmap for what needs to be done in Senior Design 2. There’s still a lot of work to do, but now that the foundation is solid, we’re confident we can build the full system and see it through to completion.

As we move forward, we’re excited not just about finishing the project, but about the opportunity to make something that works in the real world. Something that can be deployed, maintained, and used to help solve a problem that really matters. That’s what engineering is all about, and we’re proud to be part of a project that reflects that.

APPENDIX A | REFERENCES

Chatgpt, chatgpt.com/. Accessed 20 Jan. 2025.

Cass, Stephen. (2022, October 25). *Build a Passive Radar With Software-Defined Radio*. <https://spectrum.ieee.org/passive-radar-with-sdr>

Alexander Sedunov, Alexander Sutin, Nikolay Sedunov, Hady Salloum, Alexander Yakubovskiy, David Masters. (2016, December 01). *Passive acoustic system for tracking low-flying aircraft*. <https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/iet-rsn.2016.0159>

Debadeepta Dey, Christopher Geyer, Sanjiv Singh, Matt Digioia. (July 2009). *Passive, long-range detection of Aircraft: Towards a field deployable Sense and Avoid System*. <https://apps.dtic.mil/sti/pdfs/ADA518105.pdf>

SpaceX. (2023). Starlink: Satellite Internet for Remote Locations. <https://www.starlink.com>

Defense & Security Analysis. (2021). Comparison of Active and Passive Radar Systems. <https://example.com/active-vs-passive-radar>

Keysight Technologies. (n.d.). RF Monitoring Devices for Scalable Deployment. <https://www.keysight.com>

“LiTime 12V 100ah Group 24 LiFePO4 Lithium Battery.” *LiTime*, www.litime.com/products/12v-100ah-group-24-lithium-battery. Accessed 03 Mar. 2025.

Rohde & Schwarz .An Introduction to Passive Radar Systems. https://vertassets.blob.core.windows.net/download/b3c436e1/b3c436e1-3e53-4ab1-bb0c-de4dde b146eb/an_introduction_to_passive_radar_systems_wp_en_3684_0182_52_v0100.pdf

Hugh D. Griffiths, Christopher J. Baker. *An Introduction to Passive Radar*. Second Edition. Artech House: April, 2022. Print.

Krückemeier, Markus. *A Mobile Passive Radar System*. Cuvillier Verlag, 2022. eBook - PDF <https://www.perlego.com/book/3740466/a-mobile-passive-radar-system-pdf>.

“Understanding Aircraft's Communication System.” *YouTube*, uploaded by JxJ AVIATION, 6 November 2000, www.youtube.com/watch?v=iTJUHGGErp4&t=119s.

“Simple Way to Determine Aircraft Position Using a VOR, CDI, and OBS.” *YouTube*, uploaded by Loves2Fly, 20 December 2018, www.youtube.com/watch?v=jrQ21zKjIVg.

All Raspberry Pi Models and Specifications (Comparison Table). <https://raspberrypi.com/raspberry-pi-models/>

Raspberry Pi product series explained. <https://www.raspberrypi.com/news/raspberry-pi-product-series-explained/>

Qualcomm® Snapdragon™ 410E Embedded Platform.
https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/snapdragon-410e-product-brief_87-pc345-1-c_0.pdf

Digikey. *MX6 Series Processors*. <https://www.digikey.in/en/product-highlight/f/freescale/imx6>

Intel. *Datasheet Addendum: Intel Atom® Processor E3900 and A3900 Series*.
<https://www.intel.com/content/www/us/en/content-details/336256/datasheet-addendum-intel-atom-processor-e3900-and-a3900-series.html>

Verizon. https://www.verizon.com/plans/unlimited/?customer_id=279-516-8739&cmp=KNC-C-Mobility-NON-R-BPUR-NONE-NONE-2K0VZ0-COE-BIN-974&kpid=bi_cmp-416015135_adg-1235851298749322_ad-77240883752251_kwd-77240893339722:loc-4089_dev-c_ext-{extensionid}_sig-8f5caa0a9f19143b056ecdc38a8a8de8&msclkid=8f5caa0a9f19143b056ecdc38a8a8de8. Accessed 28 Feb 2025.

T-mobile. <https://www.t-mobile.com/cell-phone-plans>. Accessed 28 Feb 2025.

AT&T. <https://www.att.com/cfd/?vn=bclayout-hyper-wirelessdeals-ios&ckmid=102009&WT.srch=1&wtExtndSource=bc-offerhero-chall-d&source=EPDxATFIB00C23CFP&a=241&ocid=1023308&agid=1150090117703612&subid=2062684511.2062684511.1798399978&s1=2062684511.2062684511.1798399978&cpid=103922&gclid=51edb987a64f1660d49bde45938e5831&device=c&cookiesreset=1&c=107801&s3=e.o.&s4=c&s6=72599&ls=g&s20=103922&s19=EPDxATFIB00C23CFP&msclkid=51edb987a64f1660d49bde45938e5831>. Accessed 28 Feb 2025.

Antenna Experts. “Difference between Log Periodic Antennas and Yagi Antennas.” *Antenna Experts*, 17 Nov. 2023, www.antennaexperts.co/blog/difference-between-log-periodic-antennas-and-yagi-antennas#:~:text=Log%2Dperiodic%20dipole%20antenna%20provides,needs%20a%20narrow%20frequency%20range. Accessed 05 Mar. 2025.

Balanis, Constantine A. *Antenna Theory: Analysis and Design*. Buch Constantine A. Balanis. Wiley, 1997.

“BB60C - Real-Time Spectrum Analyzer (RTSA).” *Signal Hound*, 27 Feb. 2025, signalhound.com/products/bb60c/?srsltid=AfmBOorTBJ6kd3pw3bzsUAPbp3dIgXf6oNS43WpjXm0ezSOkWvjSOZx-. Accessed 06 Mar. 2025.

Monk, Simon. *Programming the Raspberry Pi: Getting Started with Python*. 2nd ed., McGraw-Hill Education, 2015.

Barrett, Steven F., and Daniel J. Pack. *Embedded Systems: Fundamentals with Arm Cortex-M Based Microcontrollers*. Morgan & Claypool, 2016.

Sloss, Andrew N., Dominic Symes, and Chris Wright. *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann, 2004.

Marwedel, Peter. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. 3rd ed., Springer, 2017.

Puckette, Miller. *The Theory and Technique of Electronic Music*. World Scientific Publishing, 2007.

Vacca, John R. *Computer and Information Security Handbook*. 3rd ed., Academic Press, 2017.

Krugman, Paul R., and Maurice Obstfeld. *International Economics: Theory and Policy*. 10th ed., Pearson, 2015. (*useful for cost and economic analysis of remote deployment*)

Monk, Simon. *Hacking Electronics: Learning Electronics with Arduino and Raspberry Pi*. 2nd ed., McGraw-Hill Education, 2017.

Sadiku, Matthew N.O., and Sarhan M. Musa. *Wireless Sensor Networks: Principles and Practice*. Springer, 2022.

White, Ron. *How Computers Work: The Evolution of Technology*. 10th ed., Que Publishing, 2014.

GreatScott!. “How to Work with RF 433MHz Modules | Arduino Tutorial.” *YouTube*, 15 July 2018, www.youtube.com/watch?v=9U6N8U4eMIU.

Andreas Spiess. “Raspberry Pi vs Arduino for Signal Processing? Let’s Test It!” *YouTube*, 22 Jan. 2021, www.youtube.com/watch?v=tBIoh3xopDM.

Jeff Geerling. “Raspberry Pi 4: Real-World Tests & Thermal Throttling Explained.” *YouTube*, 3 July 2019, www.youtube.com/watch?v=ppJ_FPU8_wI.

Gary Explains. “Which Is Better for Projects: Raspberry Pi or Arduino?” *YouTube*, 9 Feb. 2020, www.youtube.com/watch?v=FjObR3dpBxA.

DroneBot Workshop. “Getting Started with MQTT and Raspberry Pi.” *YouTube*, 18 June 2020, www.youtube.com/watch?v=3cubOy7vJX0.

“Lucidchart.” *Lucid Software Inc.*, www.lucidchart.com. Accessed 10 Mar. 2025.

“Draw.io – Diagrams.net.” *Diagrams.net*, www.diagrams.net. Accessed 10 Mar. 2025.

Raspberry Pi Foundation. “Raspberry Pi 4 Tech Specs.” *Raspberrypi.org*, www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. Accessed 10 Mar. 2025.

Texas Instruments. “AM335x Sitara™ Processors.” *TI.com*, www.ti.com/product/AM3358. Accessed 10 Mar. 2025.

NXP Semiconductors. “i.MX 8 Series Applications Processors.” *NXP.com*, www.nxp.com/design/software/embedded-software/i.mx-software-and-development-tools. Accessed 10 Mar. 2025.

Qualcomm. “Snapdragon Mobile Platforms.” *Qualcomm.com*, www.qualcomm.com/products/snapdragon. Accessed 10 Mar. 2025.

Intel. “Intel Atom® Processors E3900 Series.” *Intel.com*, www.intel.com/content/www/us/en/products/sku/97928/intel-atom-x7e3950-processor-2m-cache-up-to-1-60-ghz/specifications.html. Accessed 10 Mar. 2025.

Microsoft. “What Is Azure IoT?” *Microsoft Learn*, learn.microsoft.com/en-us/azure/iot-fundamentals/iot-introduction. Accessed 10 Mar. 2025.

AWS. “AWS IoT Core: Connect Internet-Enabled Devices.” *Amazon Web Services*, aws.amazon.com/iot-core/. Accessed 10 Mar. 2025.

Adafruit. “Getting Started with MQTT and Raspberry Pi.” *Adafruit Learning System*, learn.adafruit.com/adafruit-io-basics-mqtt. Accessed 10 Mar. 2025.

Particle. “What Is Edge Computing?” *Particle.io Blog*, blog.particle.io/what-is-edge-computing/. Accessed 10 Mar. 2025.

Hackster.io. “Aircraft Detection Using SDR and Raspberry Pi.” *Hackster.io*, www.hackster.io/projects/aircraft-detection-using-sdr-and-raspberry-pi. Accessed 10 Mar. 2025.

Kraus, John D., and Ronald J. Marhefka. *Antennas for All Applications*. 3rd ed., McGraw-Hill, 2001.

Balanis, Constantine A. *Antenna Theory: Analysis and Design*. 4th ed., Wiley, 2016.

Stutzman, Warren L., and Gary A. Thiele. *Antenna Theory and Design*. 3rd ed., Wiley, 2012.

Pozar, David M. *Microwave Engineering*. 4th ed., Wiley, 2011.

Hall, Peter S., and Yang Hao. *Antennas and Propagation for Body-Centric Wireless Communications*. 2nd ed., Artech House, 2012.

Lorrain, Paul, and Dale Corson. *Electromagnetic Fields and Waves*. 3rd ed., W.H. Freeman, 1988.

Collin, Robert E. *Foundations for Microwave Engineering*. 2nd ed., Wiley-IEEE Press, 2001.

Terman, Frederick E. *Radio Engineering*. 4th ed., McGraw-Hill, 1955.

Hershey, John E. *Signal Processing for Wireless Communication Systems*. Springer, 2002.

Yarbrough, John M. *Digital Logic: Applications and Design*. Cengage Learning, 2005.

“Understanding Log-Periodic Antennas.” *Microwaves & RF*, 18 Sept. 2020, www.mwrf.com/technologies/systems/article/21150234.

Turner, Ronald. “Optimizing Antenna Placement in Harsh Environments.” *IEEE Spectrum*, vol. 58, no. 3, Mar. 2021, pp. 45–49.

EEVblog. “How Spectrum Analyzers Work.” *YouTube*, 6 Jan. 2017, www.youtube.com/watch?v=ZUyAo5bjlXk.

Signal Hound. “Introduction to the BB60D Real-Time Spectrum Analyzer.” *YouTube*, Signal Hound, 12 Feb. 2018, www.youtube.com/watch?v=RceHsw-HoH0.

“Log-Periodic Antenna.” *Wikipedia*, Wikimedia Foundation, en.wikipedia.org/wiki/Log-periodic_antenna.

“Yagi–Uda Antenna.” *Wikipedia*, Wikimedia Foundation, en.wikipedia.org/wiki/Yagi%E2%80%93Uda_antenna.

“Signal Hound BB60D Datasheet.” *Signal Hound*, www.signalhound.com/signal-analyzers/bb60d.

“Starlink Mini Terminal Tech Specs.” *Starlink*, www.starlink.com/specifications.

“LPDA Antenna Basics.” *NI (formerly National Instruments)*, www.ni.com/en-us/innovations/antenna-types/log-periodic-dipole-array.html.

EEVblog. “How Spectrum Analyzers Work.” *YouTube*, 6 Jan. 2017, www.youtube.com/watch?v=ZUyAo5bjlXk.

Signal Hound. “Introduction to the BB60D Real-Time Spectrum Analyzer.” *YouTube*, 12 Feb. 2018, www.youtube.com/watch?v=RceHsw-HoH0.

GreatScott!. “How Do Antennas Work? (Basics for Beginners).” *YouTube*, 22 Apr. 2019, www.youtube.com/watch?v=0c5R77CJ2GQ.

Tech Minds. “Introduction to Passive Radar Using SDR.” *YouTube*, 16 Aug. 2022, www.youtube.com/watch?v=Wn8LCfFwFeI.

Andreas Spiess. “Raspberry Pi vs Arduino for Signal Processing? Let’s Test It!” *YouTube*, 22 Jan. 2021, www.youtube.com/watch?v=tBIoh3xopDM.

JxJ AVIATION. “Understanding Aircraft's Communication System.” *YouTube*, 6 Nov. 2000, www.youtube.com/watch?v=iTJUHGGErp4.

Loves2Fly. “Simple Way to Determine Aircraft Position Using a VOR, CDI, and OBS.” *YouTube*, 20 Dec. 2018, www.youtube.com/watch?v=jrQ21zKjIVg.

DroneBot Workshop. “Getting Started with MQTT and Raspberry Pi.” *YouTube*, 18 June 2020, www.youtube.com/watch?v=3cubOy7vJX0.

Jeff Geerling. “Raspberry Pi 4: Real-World Tests & Thermal Throttling Explained.” *YouTube*, 3 July 2019, www.youtube.com/watch?v=ppJ_FPU8_wI

Texas Instruments. TLV9051 Low-Power, Rail-to-Rail Output Op-Amp. 2023. Available at: <https://www.ti.com/product/TLV9051>

Knowles. SPU1410LR5H-QB Microphone Datasheet. 2023. Available at: <https://www.knowles.com>

Microchip Technology. MCP3008 10-Bit ADC with SPI Interface. 2023. Available at: <https://www.microchip.com>

Renogy. RNG-100D 100W Monocrystalline Solar Panel. 2023. Available at: <https://www.renogy.com>

LiTime. 12V 100Ah LiFePO4 Lithium Battery. 2023. Available at: <https://www.litime.com>

Signal Hound. BB60D Real-Time Spectrum Analyzer. 2023. Available at: <https://www.signalhound.com/products/bb60d>

Starlink. Starlink Mini Terminal Specifications. 2023. Available at: <https://www.starlink.com/specifications>

Anker. PowerCore Essential 20000 USB-C Power Bank. 2023. Available at: <https://www.anker.com>

APPENDIX B

5.1. ChatGPT vs Copilot

Prompt: “What are good aspects when using Raspberry pi model 4?”

ChatGPT’s response:

“ 1. High Processing Power: The Raspberry Pi 4 is equipped with a quad-core ARM Cortex-A72 processor (1.5 GHz), which provides the necessary computational power for real-time RF signal processing, noise filtering, and multi-threaded data handling. This makes it capable of running complex signal analysis algorithms and supporting AI-based data fusion at the edge.

2. Adequate Memory Options: Available with 2GB, 4GB, or 8GB of LPDDR4 RAM, the Pi 4 offers flexibility depending on your software demands. This is particularly important for buffering sensor data, supporting multi-process applications, or running lightweight machine learning models for aircraft classification.

3. Full Operating System Support: It can run a full Linux OS (e.g., Raspberry Pi OS), which allows you to:

Use high-level programming languages like Python, C++, and JavaScript

Leverage Linux-based tools and libraries (e.g., TensorFlow Lite, OpenCV, Paho MQTT)

Easily develop and test software in a familiar environment

This reduces development time and increases compatibility with cloud APIs, security libraries, and networking tools.

4. Rich Connectivity: Raspberry Pi 4 comes with:

Dual-band Wi-Fi (2.4GHz and 5GHz)

Gigabit Ethernet for stable wired connections

Bluetooth 5.0 for short-range wireless communication

USB 3.0 ports for fast data transfer and external sensor support

These features ensure reliable communication between distributed sensor nodes and your central system or cloud server, even in real-time.

5. **GPIO & Peripheral Interfaces:** The 40-pin GPIO header supports I²C, SPI, and UART, which are critical for interfacing with sensors (e.g., RF modules, GPS, ADCs, etc.). This enables seamless integration with your aircraft detection hardware.
6. **Scalability and Modularity:** With support for Docker, MQTT brokers, WebSockets, and cloud SDKs, the Raspberry Pi 4 allows you to scale your project easily—from a single prototype to a multi-node detection system with centralized or distributed processing.
7. **Cost-Effective and Widely Supported:** Despite its features, the Raspberry Pi 4 remains affordable and has an extensive online community. You'll find ample support, libraries, and tutorials—helping you troubleshoot and optimize your design efficiently.
8. **Support for Real-Time Enhancements:** While not a hard real-time system by default, the Pi 4 can run real-time patched Linux kernels (PREEMPT_RT) to reduce latency for more time-sensitive applications. This is ideal for real-time signal acquisition and preprocessing before cloud transmission.” (OpenAI, 2025).

Copilot response:

“The Raspberry Pi Model 4 is a versatile and cost-effective choice for building a passive aircraft detection system. Here are some of its advantages:

1. **Affordability:** The Raspberry Pi 4 is relatively inexpensive, making it accessible for hobbyists and researchers working on budget-friendly projects.
2. **Processing Power:** With its quad-core processor and up to 8GB of RAM, the Raspberry Pi 4 can handle computational tasks like signal processing and machine learning for aircraft detection.
3. **Connectivity:** It offers multiple USB ports, Gigabit Ethernet, and dual-band Wi-Fi, which are essential for connecting peripherals like software-defined radios (SDRs) and antennas.
4. **Compact Size:** Its small form factor allows for easy integration into portable or space-constrained setups.
5. **Open-Source Ecosystem:** The Raspberry Pi supports a wide range of open-source software, including tools for signal analysis and machine learning, which can be used to process aircraft signals.
6. **Community Support:** A large community of Raspberry Pi enthusiasts and developers provides extensive resources, tutorials, and troubleshooting help.
7. **Energy Efficiency:** The Raspberry Pi 4 consumes less power compared to traditional computers, making it suitable for continuous operation in remote or off-grid locations.” (Copilot, 2025)

2. What factors should I consider when selecting a Raspberry Pi for real-time signal processing?

“When choosing a Raspberry Pi model for real-time signal processing, you should consider:

Processing power: Higher clock speed and more cores allow faster computations (e.g., Raspberry Pi 4 Model B with Cortex-A72, 1.5GHz).

Memory: More RAM helps in handling large signal datasets and AI-based processing (Pi 4 offers up to 8GB).

Connectivity: USB 3.0, Gigabit Ethernet, and Wi-Fi 5 ensure fast data transfer between sensor modules and cloud systems.

Power consumption: A lower-power model like Raspberry Pi Zero 2 W is useful if the system is battery-operated.

I/O and expandability: If you need external signal processing hardware (e.g., ADC/DAC modules, DSP coprocessors), PCIe support in Compute Module 4 (CM4) is beneficial.”

3. Is the Raspberry Pi 4 Model B the best choice for real-time signal processing?

“Yes, Raspberry Pi 4 Model B is the best general-purpose choice due to:

Quad-core Cortex-A72 (1.5 GHz) offering fast processing.

Up to 8GB RAM, ensuring smooth execution of real-time data filtering and AI models.

Gigabit Ethernet and USB 3.0, useful for high-speed data transmission.

Dual micro-HDMI support, allowing real-time visualization of processed signals.

However, it consumes more power and may require active cooling for intensive tasks.”

4. When should I use a Raspberry Pi Compute Module 4 (CM4) instead?

“The Raspberry Pi Compute Module 4 is a better choice if:

You need hardware customization (e.g., industrial-grade real-time applications).

You require PCIe support to interface with high-speed DSPs or AI accelerators.

You want to embed the board into a compact system, as CM4 allows integration with custom carrier boards.

However, CM4 requires additional external components, making it less beginner-friendly than a standard Pi 4 Model B.”

5. Can Raspberry Pi Zero 2 W be used for real-time signal processing?

“Yes, but with limitations. The Raspberry Pi Zero 2 W has:

A quad-core Cortex-A53 processor (1 GHz), which is weaker than the Pi 4.

512MB RAM, limiting its ability to handle large data sets or AI-based processing.

Built-in Wi-Fi and Bluetooth, useful for lightweight wireless applications.

It is suitable for low-power applications, like basic RF signal detection or small-scale IoT systems, but not ideal for heavy real-time processing.”

6. Do I need additional hardware for real-time signal processing on Raspberry Pi?

“Yes, depending on your signal type, you may need:

Analog-to-Digital Converters (ADC) like MCP3008 or ADS1115 for analog signals.

Digital-to-Analog Converters (DAC) such as MCP4725 for signal output.

External DSP processors if you need faster real-time computations than what the Raspberry Pi CPU can handle.

Cooling solutions (fans or heatsinks), especially if performing intensive FFT analysis or AI inference on Raspberry Pi 4.”

7. Can Raspberry Pi handle AI-based real-time signal processing?

“Yes, Raspberry Pi 4 and CM4 can run lightweight AI models for signal processing, such as:

TensorFlow Lite for machine learning-based signal classification.

Edge Impulse for real-time AI-powered anomaly detection in signals.

OpenCV for image/audio-based real-time analysis.

However, for complex AI-based real-time signal processing, a dedicated edge AI processor (like Google Coral or NVIDIA Jetson Nano) may be needed.”

8. How does power consumption affect real-time processing on Raspberry Pi?

“Real-time processing increases CPU and memory usage, leading to:

Higher power consumption (Pi 4 consumes ~3W at idle but up to 7W under load).

Increased heat generation, requiring active cooling for stable performance.

Shorter battery life, if running on a battery-powered system.

For energy efficiency, consider Raspberry Pi Zero 2 W or optimize code for lower CPU usage.”

9. Can Raspberry Pi run a real-time operating system (RTOS) for better performance?

“Yes, Raspberry Pi can run RTOS-based systems like:

FreeRTOS for microcontroller-like real-time scheduling.

Preempt-RT patched Linux for low-latency real-time applications.

Xenomai for high-performance real-time Linux applications.

For hard real-time constraints, a dedicated MCU or FPGA-based system may be more suitable”.

10. What software can be used for real-time signal processing on Raspberry Pi?

“Some useful software tools include:

Python with SciPy/NumPy for real-time data filtering and FFT processing.

GNU Radio for real-time RF signal processing applications.

MATLAB & Simulink for real-time algorithm prototyping.

SoX or Audacity for real-time audio processing.

For best results, optimized C++ implementations with multi-threading are recommended”.

(OpenAI, 2025).

5.2 Case Study - ChatGPT

Prompt: “Tell me about the voltage and current requirements of our system, and how to connect our power supply unit to fill those needs”

ChatGPT’s response:

“Voltage and Current Requirements of the System & Power Supply Connections

1. Overview of Power Requirements

*Your system consists of multiple electronic components, each with specific voltage and current requirements. Since the system is deployed in a remote location, it is **solar-powered**, relying on a **12V LiFePO4 battery** and an **MPPT charge controller** to regulate power distribution.*

Below is a breakdown of the voltage and current requirements for each major subsystem:

<i>Component</i>	<i>Voltage Requirement</i>	<i>Current Requirement</i>
<i>Raspberry Pi (Data Processing Unit)</i>	<i>5V DC</i>	<i>~2.5-3A</i>
<i>Signal Hound BB60D (Spectrum Analyzer)</i>	<i>12V DC</i>	<i>~2A</i>
<i>Samsung S24 (Data Transmission)</i>	<i>5V DC (USB-C)</i>	<i>~2A</i>
<i>Low-Noise Amplifier (if used)</i>	<i>5V DC / Bias-Tee</i>	<i>~100-200mA</i>
<i>Voltage Regulators & Misc. Sensors</i>	<i>Varies (5V/3.3V)</i>	<i>~500mA - 1A</i>

LED Indicators & Monitoring Circuit	3.3V - 5V DC	~100mA
--	---------------------	---------------

2. Power Source & Connection Overview

The primary **power source** is a **12V LiFePO4 battery** charged via a **solar panel** and **MPPT charge controller**. Since not all components operate at **12V**, different **voltage regulators** are used to step down power accordingly.

- **12V Bus:** Powers the **Signal Hound BB60D** directly.
- **5V Bus:** Powers the **Raspberry Pi, Samsung S24**, and other peripherals.
- **3.3V Bus (if needed):** Powers LED indicators, sensors, and logic circuits.

3. Connecting the Power Supply to Meet System Needs

Step 1: Solar Panel to MPPT Charge Controller

1. Connect the **solar panel output** to the **PV input** of the MPPT charge controller.
2. Ensure proper **polarity (positive to positive, negative to negative)** to avoid damage.
3. The MPPT controller will optimize the charging process for the **12V LiFePO4 battery**.

Step 2: Battery Connection

1. Connect the **MPPT charge controller output** to the **12V LiFePO4 battery terminals**.
2. Use **fuse protection** (e.g., **10A fuse**) to prevent short circuits.
3. The battery stores power and provides a stable **12V DC output**.

Step 3: Voltage Regulation for Each Component

Since most components require voltages other than **12V**, **voltage regulators** are used:

12V to 5V Conversion (For Raspberry Pi, Samsung S24, LNA, etc.)

- Use a **DC-DC step-down buck converter** (e.g., **LM2596** or **MP1584**) to convert **12V** to **5V** efficiently.
- Ensure the regulator can handle at least **3A** to support the Raspberry Pi and Samsung S24.
- Connect the **5V output** to a **USB-C adapter** to charge the Samsung S24 and power the Raspberry Pi.
- If using an LNA, ensure the **5V output is clean** (minimal noise) to avoid RF interference.

Direct 12V Connection (For Signal Hound BB60D)

- *The **Signal Hound BB60D** requires a stable 12V DC supply.*
- *Use a **12V fuse (~3A)** in series to prevent overcurrent issues.*
- *Ensure proper **grounding** to minimize electrical noise.*

4. Power Distribution & Safety Considerations

To ensure stable power delivery, follow these best practices:

✓ Use proper wire gauges:

- ***Solar Panel to Charge Controller:** 10-12 AWG*
- ***Battery to Load:** 12 AWG*
- ***Voltage Regulators to Components:** 16-18 AWG*

✓ Use fuses and circuit breakers at key points:

- ***10A fuse between the battery and the charge controller***
- ***5A fuse on the 5V bus***
- ***3A fuse on the 12V line to the Signal Hound BB60D***

✓ Minimize power losses by keeping cable runs short and using efficient regulators.

✓ Implement an LED power indicator system to monitor voltage levels and detect failures.

5. Summary of Power Connections

<i>Component</i>	<i>Connected To</i>	<i>Voltage Source</i>
<i>Solar Panel</i>	<i>MPPT Charge Controller (PV Input)</i>	<i>Varies (~18V)</i>
<i>MPPT Charge Controller</i>	<i>12V LiFePO4 Battery</i>	<i>12V DC</i>
<i>Raspberry Pi</i>	<i>12V Battery → 5V Buck Converter</i>	<i>5V DC</i>
<i>Signal Hound BB60D</i>	<i>Directly to 12V Battery (Fused)</i>	<i>12V DC</i>
<i>Samsung S24 (USB-C)</i>	<i>12V Battery → 5V USB-C Adapter</i>	<i>5V DC</i>
<i>Low-Noise Amplifier (LNA)</i>	<i>5V Regulator (if used) / Bias-Tee</i>	<i>5V DC</i>

LED Indicators	3.3V/5V Regulator	3.3V - 5V DC
-----------------------	--------------------------	---------------------

By carefully managing voltage regulation and power distribution, your system will remain efficient, safe, and reliable for continuous operation in remote environments. 🚀

5.3 ChatGPT vs Grok

Chatgpt

Prompt: What kind of antenna is best to use?

Response:

The best type of antenna for your Low-Cost Passive Aircraft Detection System is a Log-Periodic Dipole Array (LPDA) antenna. Specifically, your group is already using the Planar Log-Periodic Antenna 8 from Aventas Inc., which is an excellent choice for this project. This antenna covers a wide frequency range—from 500 MHz to 8 GHz—which is ideal for picking up various types of aircraft signals such as ADS-B, VHF, UHF, and GPS. Since you're using a passive RF detection approach, having a wideband antenna ensures that you can detect signals from both cooperative and non-cooperative aircraft.

The LPDA is also highly directional, which allows for more precise detection and improves the system's ability to localize where the signal is coming from. This becomes especially useful when multiple nodes are deployed and you're using Time Difference of Arrival (TDOA) methods to triangulate aircraft positions.

Some key reasons why the LPDA is the best fit for your system:

- It provides wide frequency coverage, which is critical for detecting a range of RF signals emitted by different aircraft systems.

- It offers high directional gain, improving detection accuracy and range.

- It is scalable and modular, making it suitable for multi-node deployments like yours.

- It is weatherproof and built for outdoor, remote environments, ensuring long-term reliability.

- It strikes a good balance between performance and cost, making it a practical choice for a low-cost system.

Other antenna types, such as Yagi-Uda or dipole antennas, either lack the necessary bandwidth or offer less gain and precision. Phased array antennas are more advanced and offer electronic beam steering, but they are expensive and overly complex for your current budget and goals. Monopole and microstrip patch antennas are compact and cheap but do not provide the gain or frequency coverage needed for effective aircraft detection.

Overall, the LPDA stands out as the most suitable antenna for your project due to its performance, versatility, and durability in remote deployments.

My Response:

Why is this better than the other ones that would be recommended?

ChatGPT Response:

Why the LPDA is Better Than Other Options:

Wide Frequency Coverage

Most aircraft emit RF signals across a broad range of frequencies—VHF, UHF, ADS-B (around 1090 MHz), GPS, and more.

LPDA antennas are designed to cover a very wide frequency range (500 MHz to 8 GHz in your case), making them ideal for catching a variety of signals without needing to switch antennas.

Other antennas, like Yagi-Uda or microstrip patch, tend to work best in narrower frequency bands, which limits their usefulness if you're trying to detect multiple types of signals.

High Directional Gain

LPDA antennas are directional, meaning they focus on receiving signals from a specific area.

This increases detection range and signal quality—important when you're trying to pick up weak aircraft signals.

While dipole and monopole antennas are omnidirectional (receive from all directions), they have lower gain and are more likely to pick up background noise, making it harder to distinguish aircraft signals.

Triangulation and Localization

Your Phase II system involves multiple nodes working together to localize aircraft using Time Difference of Arrival (TDOA).

A directional antenna like an LPDA improves the precision of the time and direction of signal arrival, which is critical for accurate triangulation.

Antennas that are omnidirectional (like dipole or monopole) don't offer the same precision, which would hurt localization accuracy.

Environmental Durability

LPDA antennas (especially the Planar model you're using) are rugged and weatherproof, designed for long-term use outdoors.

Other advanced options, like phased array antennas, may offer better performance on paper but require complex electronics and are expensive, sensitive, and not ideal for harsh, remote deployments.

Cost-Effectiveness

LPDA antennas provide excellent performance without excessive cost.

Phased arrays or high-end commercial systems like Raytheon's Silent Sentry are not only costly but also complex to maintain.

LPDA gives you high gain and bandwidth while keeping the project affordable and practical.

Bottom Line:

The LPDA antenna is the most balanced solution for your system. It combines:

Wideband detection,

Directional precision,

Environmental durability,

And affordability.

Other antennas might be cheaper (like dipoles), more compact (like patches), or more advanced (like phased arrays), but none of them check all the boxes for a reliable, scalable, and low-cost passive aircraft detection system the way LPDA does.

My Response:

Could you quickly explain the installation process?

ChatGPT Response:**LPDA Antenna Installation Process (Quick Steps)****Choose the Location**

Pick a high, clear area with line-of-sight to the sky.

Avoid trees, buildings, or RF noise sources (e.g., power lines or radio towers).

Mount the Antenna

Use a metal or fiberglass mast to mount the antenna—typically on a pole or tripod.

Point the antenna in the general direction of air traffic you're interested in monitoring.

Make sure the antenna is securely fastened and weatherproofed.

Connect the Coaxial Cable

Use a low-loss coaxial cable (like LMR-400) to connect the antenna to the spectrum analyzer (Signal Hound BB60D).

If the connectors differ (e.g., N-type on the antenna, SMA on the analyzer), use the proper adapter.

(Optional) Add Signal Conditioning

For better performance, you can insert a low-noise amplifier (LNA) near the antenna to boost weak signals.

You can also use a bandpass filter to reduce out-of-band noise if you're focusing on specific aircraft signals (like ADS-B).

Connect to the Analyzer and Processor

Plug the coax cable into the Signal Hound BB60D.

The BB60D connects via USB to the Raspberry Pi or your local processor.

Power Everything On

Ensure your power system (battery and/or solar panel) is functioning.

Turn on the Raspberry Pi and run the software to start scanning RF signals.

Check Alignment and Signal Strength

Use your signal visualization software (like Signal Hound's Spike or SpectraVue) to fine-tune the antenna alignment.

Adjust as needed for maximum signal strength.

Grok:**Prompt: What kind of antenna is best to use?****Response:**

Given your project's focus on a low-cost passive aircraft detection system using multiple units arranged in an array, the choice of antenna is critical to balancing cost, performance, and practicality. Your system aims to detect aircraft signals (likely ADS-B at 1090 MHz or other RF emissions) passively, using Commercial Off-The-Shelf (COTS) components, and scale up by deploying multiple nodes to work together for triangulation and enhanced coverage. Based on your "Divide and Conquer" document and the problem statement, here's a recommendation for the best antenna type, tailored to your needs:

Recommended Antenna: Planar Log-Periodic Antenna (LPDA)

Your document already lists the Planar Log-Periodic Antenna AX-37A in the budget (priced at \$459.95 each, with 10 units planned), and it aligns well with your project's goals. Here's why it's the best choice for your low-cost, multi-unit array system:

Why a Log-Periodic Antenna?

Broadband Coverage:

LPDAs are wideband antennas, capable of receiving signals across a large frequency range (e.g., 300-3000 MHz for the AX-37A, per your specs). This ensures your system can detect ADS-B signals at 1090 MHz and potentially other aircraft RF emissions without needing multiple specialized antennas, keeping costs down.

Cost-Effectiveness:

At \$459.95 per unit, the AX-37A is a COTS component that offers good performance for the price. While not the cheapest option upfront, its broadband capability eliminates the need for multiple narrowband antennas, reducing overall system complexity and cost when scaling to an array of 10+ units.

Directional Gain:

LPDAs provide moderate gain (≥ 6 dBi per your specs), which improves signal reception over omnidirectional antennas like dipoles. This directionality is useful for an array setup, as you can orient each unit to cover specific sectors, enhancing detection accuracy when triangulating aircraft positions across multiple nodes.

Scalability for Array Deployment:

In a multi-unit array, LPDAs can be strategically placed and angled to overlap coverage zones, allowing your system to use time-difference-of-arrival (TDOA) or signal strength differences to localize aircraft. Their predictable radiation pattern supports this networked approach without requiring expensive phased-array tech.

Durability and Simplicity:

LPDAs are rugged and weather-resistant, fitting your requirement for deployment in harsh, remote environments. Their planar design (as opposed to bulky Yagi antennas) simplifies mounting and integration with your solar-powered nodes.

Comparison to Alternatives

Here's how the LPDA stacks up against other low-cost options you might consider:

Dipole Antenna:

Pros: Very cheap (often under \$50), simple to build or buy, omnidirectional.

Cons: Low gain (typically 2-3 dBi), less sensitivity to weak signals, and no directionality, making it harder to localize aircraft in an array. You'd need more nodes to achieve the same coverage.

Verdict: Too basic for your multi-unit triangulation goals.

Yagi-Uda Antenna:

Pros: Higher gain (8-12 dBi), directional, decent range for 1090 MHz if tuned properly.

Cons: Narrowband (requires specific tuning to ADS-B or other frequencies), bulkier than LPDA, and slightly more expensive per unit (around \$100-\$200 for COTS versions). Harder to adapt to varying aircraft signals.

Verdict: Good for single-frequency detection but less versatile and cost-effective than LPDA for your broadband needs.

Omnidirectional Collinear Antenna:

Pros: Affordable (\$50-\$150), 360-degree coverage, easy to deploy.

Cons: Lower gain (3-6 dBi), less precise for array-based localization since it picks up signals from all directions equally, increasing noise and false positives.

Verdict: Useful for initial detection but suboptimal for your multi-node precision goals.

Patch Antenna:

Pros: Compact, low-cost (\$20-\$100), decent gain (5-8 dBi).

Cons: Narrowband and highly directional, limiting its use to specific frequencies and requiring precise alignment, which complicates array setup in remote areas.

Verdict: Too specialized and alignment-sensitive for your scalable, low-maintenance array.

Why LPDA Wins for Your Project

Cost vs. Performance: At \$459.95 per unit, the AX-37A is pricier than a basic dipole or patch, but its broadband range and gain reduce the number of antennas and nodes needed, keeping the total system cost under your \$8,000 prototype budget when paired with other COTS parts like the Raspberry Pi and Signal Hound BB60D.

Array Synergy: Multiple LPDAs can work together effectively in your Phase 2 multi-unit network. Their directional nature allows you to cover a border area by spacing nodes 2-10 km apart (per your specs), using TDOA to pinpoint aircraft locations with your GPS-synchronized Raspberry Pis.

Low-Cost Maintenance: The LPDA's durability and lack of moving parts align with your plug-and-play, solar-powered design, minimizing upkeep in remote locations.

Practical Considerations for Your Array

Placement: Space nodes 4-10 km apart (your specs suggest 2-10 km for localization density, with a 4 km node range). Orient each LPDA to maximize coverage overlap, adjusting angles based on terrain and expected flight paths.

Integration: Connect the LPDA to the Signal Hound BB60D via a coaxial cable (SMA or N-type, as noted in your hardware design). Add a low-noise amplifier (LNA) if signals are weak, powered via a bias-tee from your 5V bus (per your power schematic).

Cost Scaling: With 10 units budgeted at \$4,599.50 total, you're within bounds for a multi-node prototype. If costs need trimming, consider fewer nodes initially (e.g., 5 units at \$2,299.75) and expand later.

Final Recommendation

Stick with the Planar Log-Periodic Antenna AX-37A as your primary antenna. It's the best low-cost, COTS option that supports your multi-unit array strategy, offering broadband detection, sufficient gain, and durability for remote deployment. It's already in your sponsor's BOM and budget, so it leverages existing resources while meeting your technical goals for passive RF detection and scalability. For your report, emphasize how its wideband capability and directional gain enable cost-effective coverage and precise localization when paired with your networked nodes.

Let me know if you'd like help refining this further for your report or calculating array placement specifics!

My Response:

So the planar log periodic antenna AX 37A is best you would say?

Grok Response:

Yes, I'd say the Planar Log-Periodic Antenna AX-37A is the best choice for your low-cost passive aircraft detection system, given your project's specific goals and constraints. Here's a concise breakdown of why it stands out:

Matches Your Requirements: It covers a wide frequency range (300-3000 MHz per your specs), including the critical 1090 MHz ADS-B signals, making it versatile for detecting various aircraft RF emissions without needing multiple antenna types.

Cost-Effective for Scale: At \$459.95 each, it's a COTS component that fits within your \$8,000 prototype budget (10 units total \$4,599.50), especially since your sponsor is already providing it. Its broadband capability reduces the need for additional hardware, keeping costs down as you scale to an array.

Array-Friendly: With a gain of ≥ 6 dBi and directional pattern, it's ideal for your multi-unit setup. Multiple AX-37As can be arranged to overlap coverage zones, enabling triangulation (via TDOA or signal strength) across your 2-10 km localization density, as outlined in your specs.

Practical for Remote Use: Its weatherproof, planar design suits your solar-powered, low-maintenance nodes in harsh environments, aligning with your 10-year durability goal.

Sponsor Alignment: It's already in your Bill of Materials (BOM) and purchased by your sponsor, so it leverages existing resources without extra funding from your team.

Compared to alternatives like dipoles (too low gain), Yagis (narrowband, bulkier), or omnidirectional antennas (less precise for localization), the AX-37A strikes the right balance of performance, affordability, and scalability for your Phase 2 multi-node network. It's not the absolute cheapest per unit, but its versatility and array synergy make it the most cost-effective overall for your system's success.

So, yes, I'd confidently recommend sticking with the AX-37A as the best antenna for your project. It's a solid foundation for your report and practical implementation. Want me to draft a paragraph for your report justifying this choice, or help with anything else related to it?

My Response:

Could you quickly explain how installation would work?

Grok Response:

Here's a quick explanation of how installation would work for the Planar Log-Periodic Antenna AX-37A in your low-cost passive aircraft detection system, focusing on setting up multiple units in an array:

Installation Process for the AX-37A in Your System

Site Selection:

Choose locations for each node based on your coverage goals (e.g., 2-10 km spacing per your specs, with a 4 km detection radius per node). Position them along a border or remote area to maximize overlap for triangulation. Consider terrain—elevated spots like hills improve line-of-sight to aircraft.

Mounting the Antenna:

Attach the AX-37A to a sturdy mast or pole (e.g., a 1-2 meter metal pole) using its mounting hardware. Its planar design is lightweight (likely under 25 kg total unit weight, per your specs), so a simple clamp or bracket works.

Orient the antenna directionally toward the expected flight path. Since it's a log-periodic antenna, it has a focused radiation pattern—angle it to cover a specific sector (e.g., 60-90° beamwidth) that overlaps with neighboring nodes.

Connecting to the Node:

Run a coaxial cable (SMA or N-type, as specified in your hardware design) from the AX-37A to the Signal Hound BB60D spectrum analyzer. Keep the cable short (e.g., 1-3 meters) to minimize signal loss.

If using a low-noise amplifier (LNA), install it near the antenna with a bias-tee powered from the 5V bus (via your Raspberry Pi or power supply), as noted in your RF detection subsystem.

Power and Hardware Setup:

Secure the node's other components (Raspberry Pi, BB60D, Samsung S24, solar panel, battery) in a weatherproof enclosure (IP65-rated, per your durability specs). Mount the solar panel above the enclosure, angled for optimal sunlight.

Connect the 12V LiFePO4 battery to the MPPT charge controller, then distribute power: 12V directly to the BB60D (via USB 3.0 from the Raspberry Pi, correcting your doc's 12V note), and 5V (via a buck converter) to the Raspberry Pi and Samsung S24.

Network Integration:

Link the Raspberry Pi to the Samsung S24 via USB tethering or Wi-Fi for Starlink SATCOM access. Ensure the Starlink Mini Terminal (or cellular service) is operational, with its antenna clear of obstructions for satellite uplink.

Testing and Calibration:

Power on the node and verify the AX-37A detects RF signals (e.g., ADS-B at 1090 MHz) through the BB60D and Raspberry Pi software. Use a known aircraft (like a commercial flight on FlightRadar24) to calibrate orientation and sensitivity.

Repeat for each node, ensuring GPS synchronization (via Raspberry Pi) for accurate TDOA across the array.

Array Coordination

Install 5-10 nodes (per your budget) in a line or grid pattern, spaced 4-10 km apart. Each AX-37A's directional coverage complements its neighbors, enabling the system to fuse data (via your software) and pinpoint aircraft locations in real-time.

Key Notes

Time: Your "plug-and-play" goal aims for setup in under an hour per node—achievable with pre-assembled enclosures and connectors.

Cost: Uses existing sponsor-provided AX-37As, minimizing additional expenses.

Scalability: Add nodes as needed, adjusting placement based on field test results.

This installation leverages the AX-37A's strengths—directional gain and broadband range—to create an effective, low-cost array. Let me know if you need a more detailed step-by-step or a diagram sketch for your report!

7.4 ERD Overview

1. Unit

Purpose: Represents the physical RF detection unit (e.g., sensors or nodes).

Attributes:

Unit_ID (PK): Primary key uniquely identifying each unit.

Unit_name: Human-readable name for identification.

Location: Geographic placement of the unit.

Status: Indicates operational status (e.g., "Active", "Offline").

Relationships:

One-to-many with RF_Signal → A single unit can detect many signals.

2. RF_Signal

Purpose: Stores raw RF data captured by each unit.

Attributes:

Signal_ID (PK): Unique ID for each signal event.

Unit_ID (FK): Foreign key linking to Unit.

Frequency: Signal frequency in MHz/GHz.

Signal_strength: Signal amplitude (dBm).

Timestamp: When the signal was captured.

Raw_data: Binary or structured format of raw signal.

Relationships:

Many-to-one with Unit.

One-to-one or one-to-many with Preprocessed_signal.

3. Preprocessed_signal

Purpose: Intermediate stage that processes raw signal data (filtering, enhancement).

Attributes:

Processed_ID (PK): Unique ID for the processed signal.

Signal_ID (FK): Links to the raw signal.

Filtered_ID: Optional field (e.g., filter profile used).

Signal_quality: Qualitative measure or score.

SNR: Signal-to-noise ratio.

Detection_flag: Boolean or integer indicating detection success (e.g., 1 = valid).

Relationships:

Many-to-one with RF_Signal.

One-to-one with Main_unit_analysis.

4. Main_unit_analysis

Purpose: Final interpretation and validation of the preprocessed signal.

Attributes:

Analysis_ID (PK): Unique analysis record.

Processed_ID (FK): Link to the preprocessed signal.

Result_status: Outcome (e.g., "Valid", "Noise", "Aircraft Detected").

Confidence_level: Percent or score (e.g., 0.95).

Timestamp: When the analysis occurred.

Analysis_notes: Any supporting comments or AI annotations.

Relationships:

One-to-one with Preprocessed_signal.

One-to-one or one-to-many with Cloud_storage.

5. Cloud_storage

Purpose: Logs final data storage to cloud for remote access, archiving, or API use.

Attributes:

Cloud_ID (PK): Unique ID for each cloud upload.

Analysis_ID (FK): Link to the final analysis that was uploaded.

Data_link: URL or identifier for stored data location.
 Storage_time: Timestamp of upload.
 API_endpoint: REST or MQTT endpoint used for access.
 Relationships:
 Many-to-one with Main_unit_analysis.

APPENDIX C - Code

Central SDR code

```
#!/usr/bin/env python3
import time, threading, queue, numpy as np, socket, json, struct
import sounddevice as sd, soundfile as sf
from collections import deque

#===== CONFIG =====
HOST_UI, PORT_UI = "127.0.0.1", 5050
TELEMETRY_PORT = 6050
AUDIO_PORT = 7050
CONTROL_PORT = 7060
AUDIO_RATE = 48_000
CHECK_INTERVAL = 1.0
JITTER_WARMUP_S = 5.0
HISTORY_SEC = 12.0
DISCONNECT_TIMEOUT = 3.0

# Expected SDRs and their Pi IPs (edit IPs as needed)
SDR_NAMES = ["SDR1", "SDR2", "SDR3"]
PI_HOSTS = {
    "SDR1": "172.20.10.3",
    "SDR2": "172.20.10.2",
    "SDR3": "172.20.10.4"
}

#===== STATE =====
current_mode = "manual" # manual | preferred | strongest
manual_sdr = "SDR1"
preferred_sdr = "SDR1"
snr_threshold = 30.0
playback_active = False

audio_queues = {n: queue.Queue(maxsize=64) for n in SDR_NAMES}
telemetry_last = {n:
{"rssi": "---", "snr": "---", "freq": 88.1, "connected": False} for n in
SDR_NAMES}
last_seen = {n: 0 for n in SDR_NAMES}
```

```

AUDIO_HISTORY = {n: deque(maxlen=int(HISTORY_SEC / 0.25)) for n in
SDR_NAMES}
state_q, cmd_q = queue.Queue(), queue.Queue()
control_conns = {} # SDR name -> live socket

#===== PROTOCOL =====
MAGIC = b"AFR1"
AUDIO_HDR = struct.Struct("!4sQII") # magic, seq(u64), nsamp(u32),
sr(u32)

#===== HELPERS =====
def build_telemetry(focused):
    tele = {"mode": current_mode, "focused": focused, "sdr": {}}
    for n in SDR_NAMES:
        d = telemetry_last.get(n, {})
        tele["sdr"][n] = {
            "rssi": d.get("rssi", "---"),
            "snr": d.get("snr", "---"),
            "freq": round(float(d.get("freq", 88.1)), 3),
            "connected": bool(d.get("connected", False))
        }
    return tele

def select_focused(tele):
    """Choose focused SDR depending on mode; ignore disconnected SDRs."""
    global manual_sdr
    sdr_data = tele["sdr"]

    if current_mode == "manual":
        return manual_sdr

    if current_mode == "preferred":
        v = sdr_data.get(preferred_sdr, {}).get("snr", -1e9)
        if isinstance(v, str): v = -1e9
        if sdr_data.get(preferred_sdr, {}).get("connected", False) and v
>= snr_threshold:
            return preferred_sdr

    def keyfn(k):
        s = sdr_data[k]
        if not s["connected"]:
            return -1e9
        snr = s["snr"]
        return snr if isinstance(snr, (int, float)) else -1e9

    try:
        return max(SDR_NAMES, key=keyfn)

```

```

except Exception:
    return manual_sdr

#===== TELEMETRY SERVER =====
def telemetry_server():
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srv.bind(("0.0.0.0", TELEMETRY_PORT)); srv.listen(8)
    print(f"[Telemetry] Listening on 0.0.0.0:{TELEMETRY_PORT}")

    def handle(conn):
        try:
            f = conn.makefile("r", encoding="utf-8", newline="\n")
            hello = json.loads(f.readline())
            name = hello.get("sdr", "UNKNOWN")
            print(f"[Telemetry] {name} connected")
            last_seen[name] = time.time()
            telemetry_last[name]["connected"] = True

            for line in f:
                try:
                    msg = json.loads(line)
                    if "sdr" not in msg:
                        continue
                    name = msg["sdr"]
                    telemetry_last[name] = {
                        "rssi": msg.get("rssi", "---"),
                        "snr": msg.get("snr", "---"),
                        "freq": msg.get("freq", 88.1),
                        "connected": bool(msg.get("connected", False))
                    }
                    last_seen[name] = time.time()
                except Exception as e:
                    print(f"[Telemetry] parse err: {e}")
            except Exception as e:
                print(f"[Telemetry] conn err: {e}")
            finally:
                conn.close()
                print(f"[Telemetry] disconnected")

        while True:
            c, _ = srv.accept()
            threading.Thread(target=handle, args=(c,), daemon=True).start()

#===== AUDIO SERVER =====
def audio_server():
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

```

```

srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
srv.bind(("0.0.0.0", AUDIO_PORT)); srv.listen(8)
print(f"[AudioRX] Listening on 0.0.0.0:{AUDIO_PORT}")

def recv_exact(sock, n):
    b = bytearray()
    while len(b) < n:
        chunk = sock.recv(n - len(b))
        if not chunk:
            raise ConnectionError("EOF")
        b.extend(chunk)
    return bytes(b)

def handle(conn):
    name = "UNKNOWN"
    try:
        f = conn.makefile("r", encoding="utf-8", newline="\n")
        hello = json.loads(f.readline())
        name = hello.get("sdr", "UNKNOWN")
        print(f"[AudioRX] {name} connected")
        last_seen[name] = time.time()
        telemetry_last[name]["connected"] = True
        f.detach()

        while True:
            hdr = recv_exact(conn, AUDIO_HDR.size)
            magic, seq, nsamp, sr = AUDIO_HDR.unpack(hdr)
            if magic != MAGIC:
                raise ValueError("Bad magic")
            payload = recv_exact(conn, nsamp * 4)
            audio = np.frombuffer(payload, dtype=np.float32,
count=nsamp)

            q = audio_queues.get(name)
            if q:
                if q.full():
                    try: q.get_nowait()
                    except: pass
                q.put((seq, sr, audio))
                last_seen[name] = time.time()
    except Exception as e:
        print(f"[AudioRX] {name} err: {e}")
    finally:
        conn.close()
        print(f"[AudioRX] {name} disconnected")

while True:
    c, _ = srv.accept()

```

```

        threading.Thread(target=handle, args=(c,), daemon=True).start()

#===== CONTROL CLIENTS (wireless set_freq) =====
def connect_control_client(host, sdr_name):
    while True:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((host, CONTROL_PORT))
            control_conns[sdr_name] = s
            print(f"[Control] Connected to {sdr_name} control at
{host}:{CONTROL_PORT}")
            return
        except Exception as e:
            #print(f"[Control] {sdr_name} connect error: {e}")
            time.sleep(3)

def send_control_command(sdr_name, obj):
    """Send a JSON line to an SDR's control connection, with one reconnect
retry."""
    msg = (json.dumps(obj) + "\n").encode()
    sock = control_conns.get(sdr_name)
    if not sock:
        print(f"[Control] No control connection for {sdr_name}")
        return False
    try:
        sock.sendall(msg)
        return True
    except Exception as e:
        print(f"[Control] send failed to {sdr_name}: {e} (retrying once)")
        # try reconnect once
        try:
            if sdr_name in PI_HOSTS:
                connect_control_client(PI_HOSTS[sdr_name], sdr_name)
                control_conns[sdr_name].sendall(msg)
                return True
        except Exception as e2:
            print(f"[Control] retry failed for {sdr_name}: {e2}")
            return False

#===== DISCONNECT MONITOR =====
def disconnect_monitor():
    was_connected = {n: False for n in SDR_NAMES}
    while True:
        now = time.time()
        for n in SDR_NAMES:
            if now - last_seen[n] > DISCONNECT_TIMEOUT:
                if was_connected.get(n, False):

```

```

        print(f"[Monitor] {n} marked disconnected (timeout)")
        telemetry_last[n]["connected"] = False
        telemetry_last[n]["snr"] = "---"
        telemetry_last[n]["rssi"] = "---"
        was_connected[n] = False
    else:
        telemetry_last[n]["connected"] = True
        was_connected[n] = True
    time.sleep(1.0)

#===== UI SOCKET =====
def start_socket_server(state_q, cmd_q):
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srv.bind((HOST_UI, PORT_UI)); srv.listen(1)
    print(f"[Server] Waiting for UI on {HOST_UI}:{PORT_UI}")
    conn, _ = srv.accept(); print("[Server] UI connected")

    def sender():
        while True:
            try:
                state = state_q.get()
                conn.sendall((json.dumps(state)+"\n").encode())
            except Exception as e:
                print(f"[Server] send error: {e}"); break

    def receiver():
        while True:
            try:
                data = conn.recv(1024)
                if not data: break
                for line in data.decode().splitlines():
                    cmd_q.put(json.loads(line))
            except Exception as e:
                print(f"[Server] recv error: {e}"); break

    threading.Thread(target=sender, daemon=True).start()
    threading.Thread(target=receiver, daemon=True).start()

#===== AUDIO PLAYER =====
def audio_player():
    global AUDIO_HISTORY, playback_active
    CHUNK_SEC = 0.25
    blocksize = int(AUDIO_RATE * CHUNK_SEC)

    stream = sd.OutputStream(channels=1, samplerate=AUDIO_RATE,
                             dtype='float32', blocksize=blocksize)

```

```

stream.start()

per_sdr_buf  = {n: {} for n in SDR_NAMES}    # seq -> audio
per_sdr_playq = {n: deque() for n in SDR_NAMES}

def feeder(name):
    next_seq = 0
    while True:
        try:
            seq, sr, audio = audio_queues[name].get()
            per_sdr_buf[name][seq] = audio
            if next_seq == 0 and seq != 0:
                next_seq = seq
            while next_seq in per_sdr_buf[name]:
                pass
        except:
            pass

per_sdr_playq[name].append(per_sdr_buf[name].pop(next_seq))
    next_seq += 1
except Exception as e:
    print(f"[Feeder {name}] err: {e}")
    time.sleep(0.05)

for n in SDR_NAMES:
    threading.Thread(target=feeder, args=(n,), daemon=True).start()

print(f"[Audio] Warming up jitter buffer for
{JITTER_WARMUP_S:.1f}s...")
time.sleep(JITTER_WARMUP_S)
print("[Audio] Jitter warmup complete.")

focused = manual_sdr
while True:
    tele = build_telemetry(focused)
    focused = select_focused(tele)
    pq = per_sdr_playq.get(focused)
    if pq and len(pq) > 0:
        chunk = pq.popleft()
    else:
        chunk = np.zeros(blocksize, dtype=np.float32)

    if playback_active:
        stream.write(np.zeros_like(chunk))
    else:
        stream.write(chunk)
        AUDIO_HISTORY[focused].append(chunk.copy())

#===== MAIN LOOP =====
def main():

```



```

elif c["cmd"] == "save_snapshot_audio":
    slot = int(c.get("slot", 1))
    try:
        name = focused
        hist = AUDIO_HISTORY.get(name, None)
        if hist and len(hist) > 0:
            data = np.concatenate(list(hist))
            filename = f"snapshot{slot}.wav"
            sf.write(filename, data, AUDIO_RATE)
            print(f"[CMD] Saved {filename}")
        else:
            print("[CMD] No audio available to save.")
    except Exception as e:
        print(f"[CMD] Snapshot save failed: {e}")

elif c["cmd"] == "play_snapshot_best":
    slot = int(c.get("slot", 1))
    filename = f"snapshot{slot}.wav"
    try:
        data, sr = sf.read(filename, dtype="float32")
        playback_active = True
        print(f"[CMD] Playing {filename} (muting live
audio)")

        sd.play(data, sr); sd.wait()
        playback_active = False
        print("[Playback] Over – resuming live audio")
    except Exception as e:
        playback_active = False
        print(f"[CMD] Playback failed for {filename}:
{e}")

# Periodic state push to UI
now = time.time()
if now - last_send >= CHECK_INTERVAL:
    tele = build_telemetry(focused)
    focused = select_focused(tele)
    tele["focused"] = focused
    state_q.put(tele)
    last_send = now

    time.sleep(0.02)
except KeyboardInterrupt:
    print("Exit")

if __name__ == "__main__":
    main()

```

Wireless SDR code

```

import time, socket, struct, json, threading, queue
import numpy as np
from rtlsdr import RtlSdr
from collections import deque
from scipy.signal import butter, sosfilt

# ===== CONFIG =====
CENTRAL_HOST = "172.20.10.14"      # <-- change to central server
IP/hostname
TELEMETRY_PORT = 6050              # shared telemetry port (all nodes)
AUDIO_PORT      = 7050              # shared audio port (all nodes)

SDR_NAME       = "SDR1"             # <-- unique per Pi: SDR1 / SDR2 / SDR3
FREQ           = 88.1e6
SDR_RATE       = 256_000
AUDIO_RATE     = 48_000
CHUNK_SIZE     = 32768              # RTL chunk
AUDIO_FRAME_SEC = 0.25              # 250 ms audio frames
TELEMETRY_HZ    = 1.0              # send 1 telemetry msg/sec
GAIN           = 'auto'

# ===== DSP =====
def lowpass_filter(audio, cutoff=15000, fs=AUDIO_RATE, order=5):
    nyq = 0.5 * fs
    sos = butter(order, cutoff/nyq, btype='low', output='sos')
    return sosfilt(sos, audio)

def fm_demodulate(samples, Fs=SDR_RATE, audio_rate=AUDIO_RATE):
    if samples is None or len(samples) < 2:
        return np.zeros(int(audio_rate * AUDIO_FRAME_SEC),
dtype=np.float32)
    phase = np.angle(samples[1:] * np.conj(samples[:-1]))
    audio = np.unwrap(phase)
    decim = int(Fs // audio_rate)
    audio_ds = audio[::decim] - np.mean(audio)
    audio_ds = lowpass_filter(audio_ds, cutoff=15000, fs=audio_rate)
    if np.max(np.abs(audio_ds)) > 0:
        audio_ds /= np.max(np.abs(audio_ds))
    return audio_ds.astype(np.float32)

def measure_fft_metrics(samples, nfft=2048):
    if samples is None or len(samples) == 0:
        return ("---", "---")
    window = np.hanning(len(samples))
    spec    = np.fft.fftshift(np.fft.fft(samples * window, nfft))
    p_db    = 10*np.log10(np.abs(spec)**2 + 1e-12)
    noise   = np.median(np.sort(p_db)[:int(0.7*len(p_db))])

```

```

    snr      = float(np.max(p_db) - noise)
    rssi     = float(10*np.log10(np.mean(10**(p_db/10))))
    return (round(rssi,2), round(snr,2))

# ===== networking helpers =====
MAGIC = b"AFR1"
AUDIO_HDR = struct.Struct("!4sQII") # magic, seq(u64), nsamples(u32),
samplerate(u32)

def send_json_lines(sock, obj):
    line = json.dumps(obj, separators=(',', ':')) + "\n"
    sock.sendall(line.encode())

def open_and_identify(host, port, sdr_name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    send_json_lines(s, {"hello": True, "sdr": sdr_name})
    return s

# ===== main =====
def main():
    # SDR init
    try:
        sdr = RtlSdr()
        sdr.sample_rate = SDR_RATE
        sdr.center_freq = FREQ
        sdr.gain         = GAIN
        print(f"[{SDR_NAME}] RTL-SDR initialized")
    except Exception as e:
        print(f"[{SDR_NAME}] SDR init failed: {e}")
        sdr = None

    # Sockets
    tele_sock = open_and_identify(CENTRAL_HOST, TELEMETRY_PORT, SDR_NAME)
    aud_sock  = open_and_identify(CENTRAL_HOST, AUDIO_PORT, SDR_NAME)

    audio_q = queue.Queue(maxsize=16)
    seq = 0
    last_tele = 0.0

    # Reader thread: grab IQ, compute telemetry, push audio frames
    def reader():
        nonlocal last_tele
        buf = np.array([], dtype=np.complex64)
        samples_per_audio_frame = int(SDR_RATE * AUDIO_FRAME_SEC)
        while True:
            try:

```

```

        if sdr is not None:
            iq = sdr.read_samples(CHUNK_SIZE) # complex64
        else:
            iq = None
        # telemetry 1 Hz
        now = time.time()
        if now - last_tele >= (1.0/TELEMETRY_HZ):
            rssi, snr = measure_fft_metrics(iq if iq is not None
else np.array([], dtype=np.complex64))
            send_json_lines(tele_sock, {
                "sdr": SDR_NAME,
                "rssi": rssi, "snr": snr,
                "freq": round((sdr.center_freq if sdr else
FREQ)/1e6, 3),
                "connected": iq is not None
            })
            last_tele = now

        # Build continuous audio
        if iq is None:
            time.sleep(0.05)
            continue
        buf = np.concatenate((buf, iq))
        while len(buf) >= samples_per_audio_frame:
            take = buf[:samples_per_audio_frame]
            buf = buf[samples_per_audio_frame:]
            audio = fm_demodulate(take, Fs=SDR_RATE,
audio_rate=AUDIO_RATE)
            # frame to queue
            if audio_q.full():
                try: audio_q.get_nowait()
                except: pass
            audio_q.put(audio)
        except Exception as e:
            print(f"[{SDR_NAME}] reader err: {e}")
            time.sleep(0.1)

threading.Thread(target=reader, daemon=True).start()

# Sender loop: pull audio frames, send with header
try:
    while True:
        audio = audio_q.get() # float32
        nsamp = audio.shape[0]
        hdr = AUDIO_HDR.pack(MAGIC, seq, nsamp, AUDIO_RATE)
        aud_sock.sendall(hdr)
        aud_sock.sendall(audio.tobytes(order="C"))

```

```

        seq += 1
    except KeyboardInterrupt:
        pass
    finally:
        try:
            if sdr: sdr.close()
        except: pass
    for s in (tele_sock, aud_sock):
        try: s.close()
        except: pass

if __name__ == "__main__":
    main()

```

UI Design code

```

import tkinter as tk
from tkinter import ttk
import socket, threading, json, time
from datetime import datetime

HOST, PORT = "127.0.0.1", 5050
UPDATE_MS = 1000

STATE = {
    "mode": "manual",
    "selected": 1,
    "sdr": {i: {"connected": False, "rssi": "---", "snr": "---", "freq": 88.1} for i in (1, 2, 3)},
}

SNAPSHOTS = [None, None]
SNAP_FRAMES = [None, None]

# ----- socket -----
def send_cmd(cmd, val):
    try:
        if client is None:
            return
        msg = (
            json.dumps({"cmd": cmd, **val}) + "\n"
            if isinstance(val, dict)
            else json.dumps({"cmd": cmd, "value": val}) + "\n"

```

```

    )
    client.sendall(msg.encode())
except Exception as e:
    print("[UI] send err", e)

```

```

def listen_backend():
    global STATE
    while True:
        try:
            data = client.recv(4096)
            if not data:
                break
            for line in data.decode().splitlines():
                try:
                    u = json.loads(line)
                    STATE["mode"] = u.get("mode", STATE["mode"])
                    f = u.get("focused", "SDR1")
                    if f.startswith("SDR"):
                        STATE["selected"] = int(f[-1])
                        for i, n in enumerate(("SDR1", "SDR2", "SDR3"), 1):
                            s = u.get("sdr", {}).get(n)
                            if s:
                                STATE["sdr"][i].update(s)
                except:
                    continue
        except Exception as e:
            print("[UI] listen err", e)
            break

```

```

try:
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((HOST, PORT))
    threading.Thread(target=listen_backend, daemon=True).start()
except Exception as e:
    print("[UI] no backend", e)
    client = None

```

```

# ----- UI -----
class App(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Frequency Diversity UI")
        self.geometry("980x580")
        self.configure(bg="#f4f5f7")
        self.option_add("*Font", "Arial 14")

        st = ttk.Style(self)
        st.configure("Card.TFrame", relief="groove", borderwidth=2, padding=10)
        st.configure("Selected.Card.TFrame", relief="groove", borderwidth=3, padding=10,
background="#d0e4ff")
        st.configure("Header.TLabel", font=("Arial", 16, "bold"))
        self.COLOR_ACTIVE = "#d0e4ff"
        self.COLOR_NEUTRAL = "#f6f7f9"

        shell = tk.Frame(self, bg="white", bd=2, relief="groove")
        shell.place(relx=0.02, rely=0.03, relwidth=0.96, relheight=0.94)

        # ===== SDR Cards =====
        self.cards_frame = ttk.Frame(shell)
        self.cards_frame.pack(fill="x", padx=16, pady=(16, 6))
        self.cards = {}
        for i in (1, 2, 3):
            f = ttk.Frame(self.cards_frame, style="Card.TFrame")
            f.pack(side="left", expand=True, fill="x", padx=8)
            title = ttk.Label(f, text=f"SDR{i}", style="Header.TLabel")
            title.pack(anchor="w")
            status = ttk.Label(f, text="● Not Connected", foreground="red")
            status.pack(anchor="w")
            snr = ttk.Label(f, text="SNR: -- dB")
            snr.pack(anchor="w")
            freq = ttk.Label(f, text="Freq: -- MHz")
            freq.pack(anchor="w")
            for w in (f, title, status, snr, freq):
                w.bind("<Button-1>", lambda e, idx=i: self.set_manual(idx))

        # Dynamic Switch Freq button
        def make_toggle_button(frm, idx):

```

```

def label_text():
    cur = STATE["sdr"][idx]["freq"]
    return (
        "Now: 88.1 → Switch to 92.6 MHz"
        if abs(cur - 88.1) < 0.05
        else "Now: 92.6 → Switch to 88.1 MHz"
    )

def toggle():
    cur = STATE["sdr"][idx]["freq"]
    new = 92.6 if abs(cur - 88.1) < 0.05 else 88.1
    send_cmd("set_freq", {"sdr": f"SDR{idx}", "value": new * 1e6})
    STATE["sdr"][idx]["freq"] = new
    btn.configure(text=label_text())
    self.refresh_cards()

    btn = tk.Button(frm, text=label_text(), font=("Arial", 12), bg="#e0e0e0",
command=toggle)
    return btn

    btn = make_toggle_button(f, i)
    btn.pack(anchor="center", pady=(4, 0))
    self.cards[i] = {"frame": f, "status": status, "snr": snr, "freq": freq}

# ===== Comparison / Live Section =====
self.compare_row = tk.Frame(shell, bg="white")
self.compare_row.pack(fill="both", expand=True, padx=16, pady=6)
self.snap_column = tk.Frame(self.compare_row, bg="white")
self.live_frame = ttk.Frame(self.compare_row, style="Card.TFrame")
self.live_frame.pack(side="right", fill="both", expand=True, padx=(10, 0))
self.live_title = ttk.Label(self.live_frame, text="LIVE FOCUS",
style="Header.TLabel")
self.live_title.pack(pady=6)
self.live_snr = ttk.Label(self.live_frame, text="SNR: -- dB", font=("Arial", 20, "bold"))
self.live_snr.pack()
self.live_rssi = ttk.Label(self.live_frame, text="RSSI: -- dBm")
self.live_rssi.pack()
self.live_freq = ttk.Label(self.live_frame, text="Freq: -- MHz")
self.live_freq.pack()
self.live_mode = ttk.Label(self.live_frame, text="Mode: MANUAL")

```

```

self.live_mode.pack()

# ===== Button Sections =====
self.controls = tk.Frame(shell, bg="white")
self.controls.pack(side="bottom", fill="x", padx=16, pady=(6, 16))

# Row 1: mode selection (3)
row1 = tk.Frame(self.controls, bg="white")
row1.pack(fill="x", pady=(0, 8))
for text, mode in [("Manual", "manual"), ("Threshold", "preferred"), ("Auto",
"strongest")]:
    b = tk.Button(
        row1,
        text=text,
        font=("Arial", 16),
        bg="#f6f7f9",
        command=lambda m=mode: self.set_mode(m),
    )
    b.pack(side="left", expand=True, fill="x", padx=8)
    if text == "Manual":
        self.btn_manual = b
    elif text == "Threshold":
        self.btn_thr = b
    else:
        self.btn_auto = b

# Row 2: snapshot + playback
row2 = tk.Frame(self.controls, bg="white")
row2.pack(fill="x")
self.btn_snap = tk.Button(
    row2, text="Snapshot", font=("Arial", 16), bg="#f6f7f9",
command=self.capture_snapshot
)
self.btn_snap.pack(side="left", expand=True, fill="x", padx=8)
self.btn_play = tk.Button(
    row2, text="Playback Best", font=("Arial", 16), bg="#f6f7f9",
command=self.playback_best
)
self.btn_play.pack(side="left", expand=True, fill="x", padx=8)

```

```

self.after(UPDATE_MS, self.tick)

# ===== SNAPSHOT HANDLERS =====
def _ensure_snap_column_packed(self):
    if not self.snap_column.winfo_ismapped():
        self.snap_column.pack(side="left", fill="both", expand=False, padx=(0, 10))

def _create_snap_frame(self, slot):
    f = ttk.Frame(self.snap_column, style="Card.TFrame")
    f.pack(fill="x", pady=(0, 10))
    close_btn = tk.Button(
        f,
        text="x",
        font=("Arial", 12, "bold"),
        bd=0,
        relief="flat",
        fg="white",
        bg="#c62828",
        activebackground="#e53935",
        command=lambda i=slot: self.clear_snapshot(i),
    )
    close_btn.place(relx=0.97, rely=0.05, anchor="ne")

    title = ttk.Label(f, text=f"SNAPSHOT {slot+1}", style="Header.TLabel")
    title.pack(anchor="w", padx=10)
    meta = ttk.Label(f, text="", anchor="w")
    meta.pack(anchor="w", padx=12)
    snr = ttk.Label(f, text="SNR: —", anchor="w")
    snr.pack(anchor="w", padx=12)
    rssi = ttk.Label(f, text="RSSI: —", anchor="w")
    rssi.pack(anchor="w", padx=12)
    freq = ttk.Label(f, text="Freq: —", anchor="w")
    freq.pack(anchor="w", padx=12)
    mode = ttk.Label(f, text="Mode: —", anchor="w")
    mode.pack(anchor="w", padx=12)

    for w in (f, title, meta, snr, rssi, freq, mode):
        w.bind("<Button-1>", lambda e, i=slot: self.play_snapshot(i + 1))

```

```

    SNAP_FRAMES[slot] = {"frame": f, "meta": meta, "snr": snr, "rssi": rssi, "freq": freq,
"mode": mode}

```

```

def _update_snap_frame(self, slot):
    d = SNAPSHOTs[slot]
    ui = SNAP_FRAMES[slot]
    if not d or not ui:
        return
    ui["meta"].configure(text=f"{d['ts']} • SDR{d['sdr']}")
    ui["snr"].configure(text=f"SNR: {d['snr']} dB")
    ui["rssi"].configure(text=f"RSSI: {d['rssi']} dBm")
    ui["freq"].configure(text=f"Freq: {d['freq']} MHz")
    ui["mode"].configure(text=f"Mode: {d['mode']}")

```

```

def clear_snapshot(self, slot):
    SNAPSHOTs[slot] = None
    if SNAP_FRAMES[slot]:
        SNAP_FRAMES[slot]["frame"].destroy()
        SNAP_FRAMES[slot] = None
    if not any(SNAPSHOTs):
        if self.snap_column.winfo_ismapped():
            self.snap_column.forget()

```

```

def capture_snapshot(self):
    sel = STATE["selected"]
    d = STATE["sdr"][sel]
    snap = {
        "ts": datetime.now().strftime("%H:%M:%S"),
        "epoch": time.time(),
        "sdr": sel,
        "snr": d.get("snr", "---"),
        "rssi": d.get("rssi", "---"),
        "freq": d.get("freq", 0.0),
        "mode": STATE["mode"].upper(),
    }

```

```

if SNAPSHOTs[0] is None:
    slot = 0
elif SNAPSHOTs[1] is None:
    slot = 1

```

```

else:
    slot = 0 if SNAPSHOT[0]["epoch"] <= SNAPSHOT[1]["epoch"] else 1

    SNAPSHOT[slot] = snap
    self._ensure_snap_column_packed()
    if SNAPSHOT[slot] is None:
        self._create_snap_frame(slot)
    self._update_snap_frame(slot)

    send_cmd("save_snapshot_audio", {"slot": slot + 1})

def playback_best(self):
    valid = [s for s in SNAPSHOT if s]
    if not valid:
        print("[UI] no snapshots")
        return
    best = max(valid, key=lambda s: s["snr"] if isinstance(s["snr"], (int, float)) else -999)
    slot = 1 if SNAPSHOT[0] == best else 2
    send_cmd("play_snapshot_best", {"slot": slot})
    print(f"[UI] Playback best snapshot{slot}.wav (SNR={best['snr']})")

def play_snapshot(self, slot):
    send_cmd("play_snapshot_best", {"slot": slot})
    print(f"[UI] Play snapshot{slot}.wav")

# ===== CORE REFRESH =====
def set_manual(self, idx):
    send_cmd("set_manual", f"SDR{idx}")
    STATE["selected"] = idx
    STATE["mode"] = "manual"
    self.set_mode("manual")

def set_mode(self, mode):
    send_cmd("set_mode", mode)
    STATE["mode"] = mode
    self._highlight_mode_button()
    self._highlight_selected_sdr()

def _highlight_mode_button(self):
    for b in (self.btn_manual, self.btn_thr, self.btn_auto):

```

```

        b.configure(bg=self.COLOR_NEUTRAL, relief="raised")
mode = STATE["mode"]
if mode == "manual":
    self.btn_manual.configure(bg=self.COLOR_ACTIVE, relief="sunken")
elif mode == "preferred":
    self.btn_thr.configure(bg=self.COLOR_ACTIVE, relief="sunken")
elif mode == "strongest":
    self.btn_auto.configure(bg=self.COLOR_ACTIVE, relief="sunken")

def _highlight_selected_sdr(self):
    selected = STATE["selected"]
    for i in (1, 2, 3):
        style = "Selected.Card.TFrame" if i == selected else "Card.TFrame"
        self.cards[i]["frame"].configure(style=style)

def refresh_cards(self):
    for i in (1, 2, 3):
        d = STATE["sdr"][i]
        c = self.cards[i]
        if d["connected"]:
            c["status"].configure(text="● Connected", foreground="green")
            c["snr"].configure(text=f"SNR: {d['snr']} dB")
            c["freq"].configure(text=f"Freq: {d['freq']} MHz")
        else:
            c["status"].configure(text="● Not Connected", foreground="red")
            c["snr"].configure(text="SNR: ---")
            c["freq"].configure(text="Freq: ---")

def refresh_live(self):
    s = STATE["selected"]
    d = STATE["sdr"][s]
    self.live_title.configure(text=f"LIVE FOCUS • SDR{s}")
    self.live_snr.configure(text=f"SNR: {d['snr']} dB")
    self.live_rssi.configure(text=f"RSSI: {d['rssi']} dBm")
    self.live_freq.configure(text=f"Freq: {d['freq']} MHz")
    self.live_mode.configure(text=f"Mode: {STATE['mode'].upper()}")

def tick(self):
    self.refresh_cards()
    self.refresh_live()

```

```
self._highlight_mode_button()  
self._highlight_selected_sdr()  
self.after(UPDATE_MS, self.tick)
```

```
if __name__ == "__main__":  
    App().mainloop()
```