Radio Diversity Automation

Kim Hoang, John Crawford, & Jurgis Siusys

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida

***Abstract* — This paper presents the design and implementation of a low-cost, software-defined radio-based system for radio diversity automation. The system demonstrates a real-time link optimization technique that dynamically selects the strongest signal from multiple spatially separated receivers. Using commercial off-the-shelf components, digital signal processing, and wireless connectivity, the platform provides a scalable proof-of-concept approach for reliable RF communication and dynamic signal detection.**

***Index Terms* — Antennas, Python, Radio Transmitters, Signal Processing Algorithms, Software Radio, Spatial Diversity**

I. INTRODUCTION

Wireless systems today depend on strong and consistent signal reception, but real-world environments often cause problems like interference, multipath fading, and signal loss. In a single-receiver setup, even small obstacles or reflections can cause the signal to drop or distort. One way to fix this issue is by using diversity reception, where multiple receivers pick up the same signal from different physical locations. When combined correctly, these separate signals can reduce fading and improve overall reliability.

The goal of this project is to build a low-cost SDR diversity system that demonstrates these concepts using common, off-the-shelf components. The design includes one transmitter and three receiver units, each with a Raspberry Pi and an SDR. The three receiver nodes are placed at different positions, allowing each to capture slightly different versions of the same transmitted signal. The data collected from each receiver is sent to a central Raspberry Pi hub, which includes a touchscreen interface. This hub acts as the control center, displaying the live data and allowing users to see how signal levels differ across each receiver.

The project's main focus is on combining hardware and software in a simple but effective way to visualize diversity reception. Each Raspberry Pi handles signal processing using Python, then shares results with the main hub using a network connection. The idea is not only to show diversity improvement in a measurable way but also to create a flexible setup that can be expanded or modified for future signal experiments.

By completing this project, we wanted to gain hands-on experience with communication systems, networking between embedded devices, and signal processing concepts often used in real wireless systems. The rest of this paper explains how the SDR diversity system was designed and built. Section II discusses how the Raspberry Pis communicate with each other. Section III covers the software and user interface design for the touchscreen hub. Later sections describe the testing process, performance results, and possible improvements for future work.

II. SYSTEM COMPONENTS

The radio diversity automation system is composed of hardware and software modules that together enable real-time RF signal comparison and selection. Each subsystem was selected for its balance of cost, performance, and ease of integration using COTS components.

A. Antenna Receivers

Three identical RF receivers are deployed at different spatial positions. Each unit captures broadcast signals and calculates signal metrics such as signal-to-noise ratio (SNR) and received signal strength indicator (RSSI)). The spatial separation allows the system to exploit environmental variations such as multipath fading and line-of-sight obstruction, enhancing overall signal resilience and diversity performance.

B. Software Defined Radios

Three identical software-defined radios interface with the receiver antennas to digitize incoming RF signals. Each SDR operates within the FM broadcast band (88–108 MHz) for proof-of-concept testing and provides real-time baseband data streams to the associated Raspberry Pi 4. This modular SDR setup enables flexible reconfiguration of frequency range, gain, and sampling rate without requiring hardware modifications.

C. Peripheral Receiving Units

Three Raspberry Pi 4s serve as peripheral controllers, aggregating data from their respective receiver units. Each performs local digital signal processing (DSP), including noise filtering, signal

strength estimation, and time-stamped metric logging. These units then forward their processed data over a local network to the central processor for diversity comparison and link selection.

D. Central Processing Unit

A Raspberry Pi 4 serves as the main controller, aggregating data from all receiver units. It performs DSP-based signal comparison, link scoring, and switching logic using a Python-based control framework. The central unit dynamically identifies the strongest and most reliable signal path and directs the system to stream data from the optimal receiver in real time.

E. Touchscreen UI

The central processor is equipped with a 3.5-inch touchscreen interface that provides live visualization of signal strength, active link status, and system control options. Through this interface, users can manually override automatic selection, choose modes of operation, and see all associated metrics.

D. Power System

Each peripheral receiver unit operates on an independent rechargeable battery pack, providing portability and flexibility during testing and field deployment. The central processing unit, responsible for coordinating all receiver data and managing the touchscreen interface, is powered through a standard AC wall connection. This configuration ensures stable operation of the main control system while maintaining mobility and modular independence for each remote receiver.

III. HARDWARE DETAIL

The following subsections describe the operation and interfacing of each subsystem as implemented within the radio diversity automation system.

A. Antenna Receivers

The antenna subsystem consists of three identical Dipole FM receivers, each designed to operate within the 88–108 MHz frequency range. Each antenna captures over-the-air broadcast signals and converts the electromagnetic waveform into a voltage signal suitable for baseband digitization by the associated Software Defined Radio.

Each receiver continuously measures key signal metrics such as Received Signal Strength Indicator (RSSI) and Signal-to-Noise Ratio (SNR). These metrics are generated from the demodulated intermediate frequency (IF) signal and provide quantifiable data for comparison across all receiver channels.

Spatial diversity is achieved by positioning each antenna receiver at different physical locations, allowing the system to mitigate multipath fading and shadowing effects. The received signal is carried through a low-loss coaxial cable to the SDR input. The cable length and impedance (50 Ω) are matched to maintain phase integrity and minimize reflection losses.

Each antenna unit is connected to its SDR via coaxial cable & an F to SMA adapter. There is a 50 to 75 ohm impedance mismatch from the antenna output to the SDR input, which results in about a 2-4dB loss. We are able to ignore this since we are getting readings of 35-40dB.

B. Software Defined Radios

The system uses three identical USB dongle-style software-defined radios (SDRs), each being the RTL-SDR Blog V4. These act as the digitization and front-end signal processing modules for the three receiver channels. Each SDR receives the analog RF output from its corresponding antenna-receiver unit and performs conversion and sampling according to the following design.

Key specification highlights (RTL-SDR Blog V4):
(1) Tuner: R828D chip.
(2) ADC/Demodulator: RTL2832U 8-bit.
(3) Frequency coverage: Approximately 500 kHz to 1.766 GHz.
(4) Typical usable bandwidth: ~2.56 MHz stable, up to ~3.2 MHz in optimal conditions.
(5) Input connector: SMA female (50 Ω) on the dongle.
(6) Improved RF front-end filtering: The V4 uses a triplexed input (HF / VHF / UHF) and built-in notch filters for common high-power broadcast interference.

Each SDR is connected via a USB 3.0 (or high-speed USB 2.0 where USB3 is unavailable) interface to a dedicated peripheral controller (a Raspberry Pi 4 board). The SDR digitally down-converts and digitizes the RF signal into a complex I/Q baseband stream, which is transferred to the Raspberry Pi for further processing.
(1) Sampling rate: The SDR units are configured to sample at a rate sufficient to capture the FM broadcast band (88–108 MHz) with 2.4–2.56 MS/s to capture the ~20 MHz span with oversampling.
(2) Gain control: Via software, each SDR's gain stage is optimized to 20, based on pre-deployment calibration and real‑time signal conditions. This can be adjusted, enabling adaptation to variable field SNR.
(3) Synchronization: While each SDR operates independently, the system enforces identical configuration parameters (sampling rate, gain range,

front-end filter settings) so that the resulting signal metric comparisons at the central processor remain consistent across channels..

The RTL-SDR Blog V4 was selected due to its strong balance of cost, performance, and ease of integration: its wide tuning ability covers the 88–108 MHz test band and leaves headroom for future expansion; the 1 ppm TCXO ensures minimal drift, which is critical when comparing receivers for diversity switching; the software-enabled bias tee offers the option of active antenna support; and the USB form factor facilitates easy connection to Raspberry Pi devices without complex hardware interfacing.

Given the ~2.56 MHz stable bandwidth limitation, capturing the entire FM broadcast band requires planning (e.g., the SDR can sweep or tune to sub-bands). In this deployment, the system focuses on a single selected program channel rather than the full band, which mitigates the bandwidth constraint. Further, while the 8-bit ADC design limits dynamic range compared to higher-end SDRs, the system's use of three spatially separated channels and signal metric comparison compensates by exploiting diversity rather than relying solely on maximal ADC fidelity.

C. Central Receiving Unit

The central controller for the radio diversity automation system is implemented on a Raspberry Pi 4 Model B equipped with 4 GB of RAM. It serves as the master coordination node, aggregating signal data from the three remote receiving units over a local Ethernet or Wi-Fi network. Its primary functions include digital signal comparison, diversity decision logic, and control interface management through an integrated graphical user interface (GUI).

The backend server is written in Python 3, using the standard libraries socket, threading, and json to maintain persistent TCP connections with each peripheral Raspberry Pi. Each peripheral transmits JSON-encoded telemetry containing the following fields:

(1) RSSI (dBm) – derived signal power level
(2) SNR (dB) – instantaneous signal-to-noise ratio estimate
(3) Frequency (MHz) – tuned frequency of the SDR
(4) Connection status and timestamp

The central unit receives and buffers these metrics in real time, performing statistical comparisons to determine link quality. The system's link selection algorithm ranks the three inputs according to signal stability and strength, identifying the optimal channel for continuous streaming.

The backend continuously updates a shared global state dictionary and sends it to the user interface thread once per second. This state synchronization is performed using thread-safe operations to prevent data collision between the socket and UI threads.

Three selectable operating modes govern system behavior:

(1) Manual Mode: The user explicitly chooses which SDR to monitor, overriding automated switching.
(2) Preferred Mode: The system tracks a user-selected "preferred" receiver unless its quality falls below a defined SNR threshold.
(3) Strongest Mode: The controller dynamically switches to the receiver exhibiting the highest SNR and RSSI at any given time.

These control states are broadcast as JSON messages over the same TCP channel to the peripheral Pis, which adjust their data routing accordingly. Switching latency between receivers averages under 500 ms, limited primarily by network buffer time and the SDR sample window.

The 3.5-inch TFT touchscreen connected via GPIO displays a Tkinter-based GUI built atop the Python tkinter and ttk libraries. The interface provides real-time visualization of all three SDR channels and implements full manual and automatic control through an event-driven model.

Each SDR is represented by an independent status card displaying:

(1) Connection state (color-coded LED indicator)
(2) Signal-to-Noise Ratio (SNR)
(3) Tuned frequency
(4) Real-time updates at 1 Hz refresh rate

A focused view panel displays detailed metrics of the currently selected SDR, including RSSI, SNR, and frequency. Three mode buttons (Manual, Preferred, and Strongest) allow the operator to toggle between system states. Button presses trigger the send_cmd() function, which transmits JSON control messages to the backend process in the format:

{"cmd":"set_mode","value":"strongest"})

The GUI's internal update loop operates through the Tkinter after() method, invoking the tick() function once per second to refresh displayed data. The interface runs as a non-blocking process, ensuring continuous operation of the backend communication thread.

The flow of communication is as follows:

(1) Peripheral Pis compute and transmit RSSI/SNR telemetry via TCP (port 5050).
(2) The central backend receives and parses the JSON stream.
(3) The backend updates a shared state object representing all SDRs.
(4) The Tkinter GUI thread periodically polls this state and updates on-screen metrics.
(5) User interactions (button presses or SDR selection) send JSON control commands back to the

backend, which propagates them to the peripheral units.

This architecture enables real-time bidirectional control between the operator and the distributed SDR network while maintaining robust separation between data acquisition, processing, and user interaction layers. The use of JSON and socket-based communication ensures modular extensibility, so additional receivers or alternate visualization platforms can be added with minimal code modification.

IV. SOFTWARE CONCEPT

The software flowchart illustrated in Fig. 1 demonstrates the operational logic of the proposed radio diversity detection system.

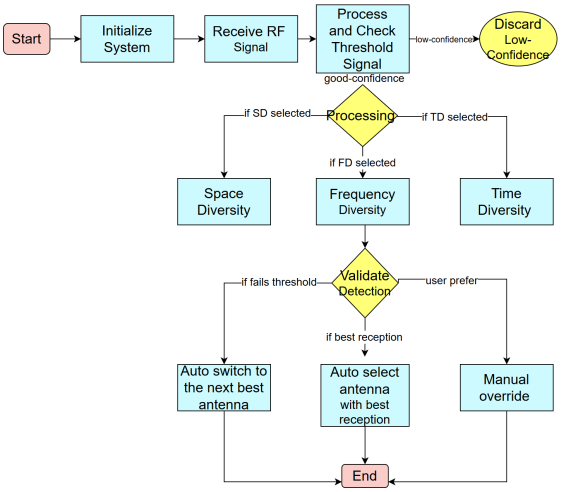

Fig 1. Software flowchart

The process begins with the initialization stage, during which all hardware components—including antennas, RF receivers, and processing modules—are configured and made ready for operation. Once initialization is completed, the system proceeds to receive the incoming RF signal. This signal is then forwarded to the signal processing and threshold verification module, where its amplitude and quality are analyzed against predefined confidence thresholds. Signals that fail to meet the minimum confidence level are categorized as low-confidence and subsequently discarded to prevent false detections. In contrast, signals that exceed the confidence threshold are labeled as good-confidence and proceed to the diversity processing stage.

At this stage, the system determines the diversity method selected by the user or automatically configured by the control software. Depending on the operational mode, the algorithm engages one of three diversity techniques: Space Diversity (SD), Frequency Diversity (FD), or Time Diversity (TD). In the Space Diversity mode, multiple antennas are evaluated based on their respective signal-to-noise ratios (SNRs) to identify the antenna providing the highest-quality reception. In the Frequency Diversity mode, the system compares multiple frequency channels to identify the one with optimal performance, mitigating the impact of frequency-selective fading. Meanwhile, in the Time Diversity mode, signal samples collected at different time intervals are averaged or combined to enhance detection reliability under varying environmental conditions.

Following the diversity operation, the system enters the validation stage, where the detection outcome is analyzed to ensure that it satisfies performance thresholds or user-defined parameters. If the validated signal fails to meet the required threshold, the algorithm automatically switches to the next best antenna to maintain continuous signal tracking and robustness. Conversely, if the detection is validated and offers the best reception, the software automatically selects that antenna as the active input source. A manual override function is also incorporated, enabling users to intervene and manually select the preferred antenna configuration when desired. Once the optimal reception condition is established—either through automatic selection or manual adjustment—the process concludes, marking the end of the software operation cycle.

V. SOFTWARE DETAIL

A. Frontend: User Interface (UI) Development and Telemetry Visualization

The graphical user interface for the radio diversity detection system was developed in Python using the Tkinter library, chosen for its lightweight integration with the Raspberry Pi 4 environment. This interface provides real-time visualization of key telemetry metrics such as Signal-to-Noise Ratio (SNR), Received Signal Strength Indicator (RSSI), operating frequency, and connection status for up to three connected RTL-SDR receivers (labeled SDR1, SDR2, and SDR3). These data are displayed on individual “cards” for each SDR, providing a clear and interactive layout for users to monitor and evaluate performance.

The UI supports three operating modes—Manual, Preferred, and Strongest—which the user can select via the interface. In manual mode, users select a fixed SDR input for signal monitoring. The preferred mode allows the system to use a user-specified SDR, as long as its signal quality meets a predefined SNR threshold. The strongest mode, by contrast, continuously evaluates all SDR inputs and

automatically selects the one with the best signal reception.
To enable communication between the UI and backend signal processing module, the system uses a TCP socket connection. The UI acts as a TCP client, receiving telemetry data in JSON format and sending mode-switching commands to the backend:

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((HOST, PORT))
```

Each second, the UI receives telemetry packets and updates the global STATE dictionary that holds metadata for each SDR, including connection status and signal metrics. Example structure:

```
STATE = {
  "mode": "manual",
  "selected": 1,
  "sdr": {
    1: {"connected": False, "rssi": "---",
"snr": "---", "freq": 0.0, "age": 0.0},
    2: {"connected": False, "rssi": "---",
"snr": "---", "freq": 0.0, "age": 0.0},
    3: {"connected": False, "rssi": "---",
"snr": "---", "freq": 0.0, "age": 0.0}
  }
}
```

Real-time telemetry updates are parsed and assigned using:

```
update = json.loads(line)
STATE["sdr"][i].update({
  "rssi": d.get("rssi", "---"),
  "snr": d.get("snr", "---"),
  "freq": d.get("freq", 0.0),
  "connected": d.get("connected", False),
})
```

The user interface refreshes at a 1-second interval, giving users a near real-time view of system behavior.

B. Backend: RTL-SDR Signal Processing and Diversity Control Logic

The backend component, also written in Python, operates entirely on the Raspberry Pi 4, without requiring a separate computer. It manages low-level signal processing and diversity logic by interfacing with multiple RTL-SDR USB dongles, each initialized with individual gain and frequency settings. The backend uses the pyrtlsdr library to collect complex I/Q samples, from which it derives critical signal quality metrics such as RSSI and SNR.

For each SDR, a dedicated thread is launched to independently read samples, enabling parallel data acquisition:

```
threading.Thread(target=sdr_reader, args=(sdr, sample_queues[name], name),
 daemon=True).start()
```

The signal quality is computed using functions like the following:

```
def measure_snr(samples):
  power = np.mean(np.abs(samples)**2)
  noise_floor = np.median(np.abs(samples)**2)
  snr_linear = power / noise_floor if noise_floor > 0 else
np.inf
  return 10 * np.log10(snr_linear)
```

This backend supports the three operating modes described earlier:
(1) Strongest Mode: Continuously selects the SDR with the highest measured SNR.
(2) Preferred Mode: Uses a designated SDR if it satisfies the SNR threshold, otherwise switches to a better one.
(3) Manual Mode: User manually selects the SDR to be used regardless of signal strength.
After each evaluation cycle, the backend compiles telemetry data into a JSON object that is sent to the frontend:

```
telemetry["sdr"][name] = {
  "rssi": round(rssi, 2),
  "snr": round(snr, 2),
  "freq": round(freq, 3),
  "connected": connected
}
```

The diversity decision logic is driven by a custom flowchart, designed to reflect real-world RF techniques such as space diversity (evaluating multiple antennas), frequency diversity (scanning across different channels), and time diversity (temporal comparisons for fluctuating signals).

C. Real-Time System Integration

By combining a responsive user interface with a multi-threaded backend running on a single-board computer, the system achieves fully autonomous radio signal diversity detection. This setup is particularly useful for applications in wireless sensing, passive surveillance, or resilient RF communications in low-cost embedded systems. The Raspberry Pi 4 proves to be a capable platform for real-time signal monitoring without the need for external computation resources.

VI. COMMUNICATIONS OVERVIEW

The communication system is a key part of making the SDR diversity setup function as one connected unit. Each receiver node collects its own signal data, then sends that data to the main Raspberry Pi hub so that all results can be viewed and compared. To make this possible, a TCP/IP network connection is used between the Pis. Each receiver runs a Python program that calculates key values such as signal strength (RSSI) or signal-to-noise ratio (SNR) from the SDR input. These values are then sent to the main hub using small, structured packets that include the node's ID and a timestamp.

The main Raspberry Pi acts as a server, while each receiver acts as a client that connects to it. Once connected, each client keeps sending new data continuously while the system is running. On the hub side, another Python script receives these packets on separate network ports, stores them, and updates the user interface in real time. This setup allows all three receivers to operate at the same time without interfering with each other.

One major advantage of this approach is that it avoids sending large amounts of raw SDR data across the network. Instead, only the important measurements are sent, which keeps the system fast and responsive. The data packets are small and easy to parse, which makes it possible to update the display on the hub almost instantly. This also keeps the CPU and network usage low, allowing the system to run smoothly even on Raspberry Pi hardware.

The communication layer uses multithreading so that each connection can be handled independently. This prevents the hub from freezing or slowing down if one of the receiver nodes experiences a delay. Each receiver also sends a short "heartbeat" message every few seconds to let the hub know it's still connected. If the hub stops receiving updates from one of the nodes, it flags that node as disconnected so the user knows there's a problem.

The nodes can communicate either through Ethernet for a wired lab setup or through Wi-Fi for a more flexible layout. In our lab tests, Ethernet provided the most stable results, while Wi-Fi made it easier to spread out the receivers for better spatial separation. In future versions of the project, the team hopes to make the receivers communicate wirelessly using a mesh network, allowing them to be placed farther apart in the field.

Overall, the communication network forms the backbone of the SDR diversity system. It keeps all nodes synchronized and ensures that signal data is collected and displayed correctly. Without this reliable data link, the system would not be able to show how spatial diversity improves signal reception in real time.

VII. GRAPHICAL USER INTERFACE

The graphical user interface (GUI) is one of the most important parts of our project because it's what brings everything together visually. It runs on the main Raspberry Pi, which also acts as the central hub that collects all the data coming from the three receiver Pis. The goal of the GUI is to take the signal data sent by each receiver and display it in a way that's easy to understand and track in real time. Since the main Pi is connected to a 7-inch touchscreen, the whole system can be controlled without a keyboard or mouse, making it compact and simple to use during testing.

A. Design Goals

When we first started planning the interface, our goal was to make something that looked clean, updated smoothly, and gave us a quick way to see what was happening with the system. We wanted the GUI to:

(1) Show real-time signal readings from each receiver.

(2) Be easy to use and not require command-line input.

(3) Give clear status feedback if a receiver disconnected or stopped sending data.

(4) Be scalable so more receivers could be added later without rebuilding the whole thing.

We decided to build it in Python using the Tkinter library because it's already included with Raspberry Pi OS and works well with other Python code. Tkinter gave us a lot of flexibility for laying out labels, buttons, and graphs without being too heavy for the Pi to run.

B. How it Works

The GUI is basically the "front end" of the system. Under the hood, it connects to the three receiver Pis using TCP/IP sockets and listens for data packets they send every few seconds. Each receiver sends things like signal strength (RSSI), noise level, and a timestamp. When the hub Pi receives that data, it updates the screen so you can instantly see which receiver has the best signal and how they compare. We set up the code so that each receiver runs in its own thread, meaning all three can send and update data at the same time without freezing the screen. Another thread runs the GUI itself, refreshing the numbers and small graphs every few hundred milliseconds. This setup keeps the interface responsive even when data is coming in constantly.

C. Layout and Display

The main screen is divided into sections for each receiver. At the top of the GUI, there's a header with the project name, system time, and network status. Below that, each receiver has its own panel showing:
(1) Receiver ID (e.g., "Receiver 1") Signal strength and SNR readings
(2) A small scrolling graph showing how the signal changes over time
(3) A colored status box (green = connected, red = disconnected)

On the right side, we added simple control buttons *Start*, *Stop*, and *Reset*. These let us start data collection, pause updates, or clear the display without having to restart the program. At the bottom of the screen, there's a larger graph that combines all three receivers' readings so we can see diversity performance more clearly. At the bottom of the display, there's a summary area that gives an overview of the system's current state, showing which receivers are active and confirming that data is being received from each one. The interface focuses on providing clear visual feedback and smooth interaction, making it simple for the user to monitor and control the setup without dealing with command lines or technical menus. This approach keeps the system lightweight and easy to use while running reliably on the Raspberry Pi's limited hardware.

D. Data Handling and Updates

Each receiver sends its data in a small, easy-to-read format that includes its ID, signal values, and a timestamp. The hub Pi parses this information and stores it in a local list or buffer. The GUI then updates each section of the screen with the latest readings. If one of the receivers stops sending data, the GUI automatically marks it as disconnected. This helps us quickly see if a Pi or SDR is having issues.

We also added a simple data logging feature that saves all the incoming readings into CSV files. That way, after we're done testing, we can look back and analyze how signal strength or noise levels changed during the experiment.

E. Development Challenges

Making the GUI run smoothly on the Raspberry Pi wasn't easy at first. Since the Pi isn't very powerful, updating several graphs in real time caused the program to lag. We fixed this by lowering the refresh rate slightly and optimizing how often we redraw the plots. Another issue was keeping the communication stable. Sometimes, if the Wi-Fi connection dropped for a second, it would cause the GUI to freeze. We solved this by adding short timeouts and a "heartbeat" signal from each receiver so the hub knows whether it's still connected.

Integrating all of the code was probably the most time-consuming part. We had to make sure the networking code, data processing, and Tkinter updates didn't interfere with each other. Splitting things into separate threads and keeping the functions modular made the system more stable and easier to debug.

F. Future Improvements

In the future, we'd like to make the GUI look a little more polished and maybe use a different framework like PyQt for a smoother interface. It would also be nice to add some extra features, like showing graphs of all receivers over time on one screen or even controlling the receivers remotely from the main Pi. Another idea is to host a small web server on the hub so we could view the data from a laptop or phone instead of just the touchscreen.

G. Summary

Overall, the GUI turned out to be one of the most satisfying parts of the project. It takes all the background work happening with the SDRs and network communication and puts it into a form that's easy to understand and visually interesting. It also made testing much easier since we could see everything happening live instead of just reading numbers in a terminal. The interface made our SDR diversity system feel more like a real, complete product rather than just a technical experiment.

VIII. PCB DESIGN

Although the original senior design requirements included the development of a printed circuit board (PCB), the PCB requirement was formally waived for this project due to its nature as a software-defined and modular prototype. The system's primary objectives—radio diversity analysis, signal quality comparison, and automated link selection—were best demonstrated using commercial off-the-shelf (COTS) SDR hardware, Raspberry Pi units, and modular interconnects. Since the core functionality resides in the software and digital signal processing algorithms rather than in a fixed analog circuit, a PCB was not essential to validate the design or meet the proof-of-concept goals.

A. Integration-Based PCB Option

If the project were to advance toward a field-deployable or production variant, a PCB could be implemented to integrate the SDR front-ends, power regulation, and analog filtering into a single compact enclosure. This integration would reduce wiring complexity, improve signal integrity, and increase environmental robustness.

B. Power and Control PCB Option

Another potential use for a PCB would be in the power distribution and system control network. A custom board could consolidate solar input, MPPT charge control, battery management, and sensor telemetry into a unified system interface. This would streamline assembly, reduce failure points, and allow for embedded monitoring circuits such as voltage and current sensing.

C. Embedded System PCB Option

Finally, for future versions where the diversity logic is embedded directly into microcontrollers or system-on-module hardware, a dedicated PCB could host both RF front-end components and a microprocessor. This approach would support scalability, miniaturization, and long-term deployment in rugged outdoor environments.

THE ENGINEERS

Kim Hoang is a Computer Engineering student with experience as a Product Engineering Intern at Leonardo DRS. Kim enjoys hands-on work involving embedded hardware and software and aspires to work for leading defense companies such as Lockheed Martin, L3Harris, and Leonardo DRS.

John Crawford is a 22-year old Electrical Engineering student. John hopes to pursue a job in circuit & analog filter design or digital signal processing, specifically with electronics geared towards audio & sound engineering.

Jurgis Siusys is a Computer Engineering student at the University of Central Florida. Jurgis focuses on the electrical and hardware side of the degree, with a strong interest in circuit design, embedded systems, and hardware development. He hopes to pursue a career in hardware engineering or circuit design.

REFERENCES

RTL-SDR Blog: A Guide to Cheap Software Defined Radio. Carlson, Carl. RTL-SDR Tutorial: Cheap Software Defined Radio. RTL-SDR Blog, https://www.rtl-sdr.com/about-rtl-sdr/

Raspberry Pi Foundation. “Raspberry Pi 4 Tech Specs.” *Raspberry Pi*, 2023, https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

RTL-SDR Blog: A Guide to Cheap Software Defined Radio. Carlson, Carl. RTL-SDR Tutorial: Cheap Software Defined Radio. RTL-SDR Blog, https://www.rtl-sdr.com/about-rtl-sdr/