NEXASENSEE

Next Generation Environmental Sensing

EE40190 Senior Design II

Final Report

May 7th, 2025

Name	NDID
Jeff M. Mwathi	902182754
Katherine Davila	902167325
AnnahMarie Behn	902172819
Kyle Crean	902180910
Jeffrey Yang	902179486

TABLE OF CONTENTS

Introduction

Motivation and Problem Statement

Electromagnetic interference (EMI) poses a significant threat to the reliability and safety of electronic systems operating in electromagnetically sensitive environments. EMI arises from both internal and external sources; including wireless communication devices (e.g., mobile phones, Wi-Fi, Bluetooth), switching power supplies, high-voltage transmission lines, and natural phenomena such as lightning or solar activity [1]. In high-stakes settings such as intensive care units (ICUs), laboratories, cleanrooms, and precision manufacturing facilities, even brief episodes of EMI can lead to data corruption, device malfunction, or system failure. To mitigate these risks, global regulatory frameworks such as FCC Part 15 in the United States and the European EMC Directive have established strict emission limits to ensure electromagnetic compatibility (EMC) and safe coexistence among devices [2].

Even when devices comply with regulatory standards, recent studies reveal that EMI remains a persistent concern in clinical settings. In an investigation published by the National Library of Medicine, researchers conducted over 5,800 electromagnetic field measurements within a hospital ICU, covering the 47 MHz - 2.5 GHz spectrum [3]. Although all measurements were below legal exposure limits, peak field strengths reached 3.55 V/m, with digital enhanced cordless telecommunications (DECT) devices and mobile phones contributing more than 65% of the total radiofrequency (RF) exposure. Notably, one referenced clinical study reported that 3 out of 22 ventilators exhibited shutdowns or abnormal behavior when exposed to RF emissions from proximate mobile phones. A broader clinical review identified over 40 documented EMI-related incidents, including infusion pump errors, false readings, and complete ventilator failure; illustrating the continued vulnerability of life-sustaining equipment to RF interference.

The growing density of wireless systems and the increasing interconnectivity of modern devices have intensified the difficulty of ensuring EMI-safe operation in environments where precision and safety are priority. This issue is not limited to healthcare; it extends to any setting where EMI can compromise data fidelity or disrupt mission-critical operations.

To address this challenge, we present a modular environmental monitoring system that utilizes free-space optical (FSO) communication to avoid RF emissions during data transmission. By leveraging light-based communication in lieu of RF signaling, the proposed system enables real-time environmental sensing while maintaining RF silence. This design is particularly advantageous for deployment in ICUs, cleanrooms, and other EMI-sensitive domains, where electromagnetic compatibility, operational integrity, and low-power performance are essential.

Evaluation of Design Outcomes

Our final system successfully met the majority of our core design objectives and subsystem-level requirements, particularly in the areas of sensing, optical transmission, data integrity, and user interface integration. Despite constraints on time and subsystem complexity, the prototype demonstrated robust functionality under real-world testing and evaluation.

Overview of the proposed solution

The system consists of a battery-powered transmitter hub and a USB-powered receiver hub that communicate via a custom infrared optical link. Environmental data is collected at the transmitter, encoded with (8,4) Hamming error correction, and sent using IR pulses. The receiver captures these signals through a photodiode, reconstructs the original data, and sends it via Ethernet to a central console. A custom GUI on the console displays real-time data and manages logging and user interaction. The modular components are housed in compact, mountable enclosures that support line-of-sight alignment and reduce installation complexity.

Subsystem Overview

Sensor Subsystem

The transmitter hub integrates a multi-sensor suite consisting of the BME680 (temperature, humidity, pressure, gas), BH1750 (ambient light), and SPH0645LM4H (sound). Sensor data is sampled periodically, time-stamped, and formatted by the ESP32-S3 microcontroller, which also manages low-level I²C and I²S communication protocols.

Optical Communication Subsystem

Encoded sensor data is transmitted optically using an infrared LED (TSAL6200) modulated via UART, implementing either On-Off Keying (OOK) or the IrDA protocol at 115.2 kbps. At the receiver hub, a BPW34 photodiode detects the signal, which is amplified by a high-gain transimpedance amplifier (OPA657) and thresholded by a high-speed comparator (TLV3501) to restore a clean digital waveform.

Power Subsystem

The transmitter is powered by a 3.7V lithium-ion battery, stepped down to 3.3V using a TPS63802 buck converter. The receiver hub is powered through a USB-C interface, regulated via an AZ1117-3.3V LDO and supported by an LM27762 charge pump that provides ±3V rails for dual-supply analog components.

Data Processing and Integration

The ESP32-S3 on both hubs manages data formatting, optical signal processing, and error correction using (8,4) Hamming codes. On the receiver side, the microcontroller transmits decoded packets via SPI to a W5500 Ethernet controller, enabling reliable, low-latency communication with the central console. The console GUI allows for real-time visualization, logging, and data export.

Detailed System Requirements

The following system-level requirements were established to guide the design and implementation of a free-space optical environmental monitoring solution tailored for electromagnetic interference (EMI)-sensitive environments. These specifications address functionality, performance, mechanical form factor, communication protocols, and expandability.

Functional Requirements

The system shall perform real-time environmental monitoring while ensuring zero RF emissions. It must:

- Measure environmental parameters with precision, including:
    - Temperature (°C)
    - Humidity (% RH)
    - Atmospheric pressure (hPa)
    - Gas concentration (kΩ)
    - Ambient light intensity (lux)
    - Sound level (dB)
- Transmit data optically using infrared modulation
- Receive, decode, and forward data to a central console via Ethernet
- Display data in a real-time graphical user interface (GUI) with logging capabilities

Performance Requirements

Quantitative performance benchmarks were established based on real-time responsiveness and EMI-safety constraints:

- Transmission rate: $\geq\ 100\ kbps$ (OOK, 115.2 kbps demonstrated)
- Transmission range: $\geq\ 3\ meters$ (line of sight)
- Data Latency: $\leq\ 1\ second$ from sensing to GUI display
- Sensor tolerances: Temp = $\pm\ 1^{\circ}C$; Humidity = $\pm\ 3\%\ RH$; Pressure $\pm\ 1hPa$
- Active Current Draw: $\leq\ 150\ mA$
- Minimum Battery lifetime: $\geq\ 2\ weeks$

EMI Compliance and Optical Communication Requirements

Given the use case in EMI-critical spaces, all data transmission must occur via optical links:

- Communication shall be limited to the infrared spectrum
- The system shall not emit RF signals
- Data encoding shall be applied to transmitted data to support error correction
- Analog conditioning of photodiode signals shall preserve signal fidelity in the presence of ambient light and electronic noise.

Power and Energy Requirements

- Transmitter Hub
    - Power source: 3.7V lithium-ion battery.
    - Regulation: TPS63802 buck converter for 3.3V rail.
    - Hardware must support future sleep mode operation for battery optimization.
- Receiver Hub
    - Power source: USB-C (5V).
    - Regulation: AZ1117 LDO for 3.3V; LM27762 charge pump for ±3V analog rails.

Enclosure Design Requirements

The system's housing was designed to support stable operation, signal accessibility, and safe deployment in wall-mounted configurations. The following criteria governed the mechanical design:

1. The housing must securely hold the PCB and associated components, with internal clearance for connectors and routing.
2. The transmitter hub must include a dedicated space to safely store the 3.7V lithium-ion battery, ensuring mechanical stability and thermal dissipation. Additionally, physical access to critical ports: Ethernet, USB-C, battery JST connector, and debugging headers.
3. The enclosures must maintain line-of-sight between the infrared LED and photodiode, requiring openings in the lid to expose these components for reliable optical communication.
4. Specific sensors, such as the ambient light sensor (BH1750), require exposure to ambient conditions, leading to designed openings or clear sections in the housing.
5. For accurate gas and temperature/humidity sensing, the enclosure included ventilation slots around the side walls and lid. These served to equalize interior and ambient air and provide passive cooling to avoid thermal buildup.
6. Mounting compatibility with vertical surfaces (e.g., adhesive or screw-based) was prioritized. As such, the enclosure was designed to be lightweight and low-profile to avoid load-related detachment and to minimize visual impact in clean environments.

Interface and Communication Protocols

These interfaces define how sensor data is collected, processed, transmitted, and received, while supporting power delivery and system debugging. Each interface was chosen based on bandwidth needs, peripheral compatibility, and protocol simplicity.

- I^2C = Sensor integration (BME680, BH1750)
- I^2S = Digital Microphone input (SPH0645LM4H)
- UART = optical modulation control and data transmission
- GPIO + ADC = Photodiode signal capture and comparator thresholding
- SPI = Ethernet Communication (ESP32 → W5500)

- USB-C = power delivery and debug interface

GUI and Data Logging Requirements

- Real-time visualization of sensor data.
- Logging/export of data (.csv or .xlsx format).
- Threshold alerting for key metrics.
- End-to-end update latency ≤ 1 second.

These system-level requirements ensured alignment between the functional goals and subsystem-level implementations and formed the benchmark against which final prototype performance was assessed.

Detailed Project Description

System Theory of Operation

The complete system is a modular, free-space optical sensor network designed for electromagnetic interference-sensitive environments. It consists of three primary components: a battery-powered transmitter hub, a wall-mounted receiver hub, and a central desktop console. Each module serves a distinct function within a continuous data pipeline, from sensing to user interface.

The transmitter hub collects environmental data from onboard sensors, including temperature, humidity, atmospheric pressure, gas concentration, ambient light intensity, and sound level. These sensors communicate with the ESP32-S3 microcontroller over I2C and I2S protocols. Sensor readings are packaged into a structured data string, encoded using a Hamming (8,4) algorithm for error detection and single-bit correction, then transmitted as modulated infrared pulses through a 940 nm LED. The optical transmission occurs at a fixed rate of 1 Hz and maintains a line-of-sight range of approximately 3 meters.

On the receiver hub, a photodiode circuit captures the incoming infrared signal and converts it to an electrical waveform. This signal is fed into the ESP32-S3 through a UART interface, where it is demodulated and decoded using a software-based Hamming decoder. Once reconstructed, the data string is transmitted over Ethernet using the W5500 chip via the UDP protocol. The receiver firmware defines a fixed MAC address and uses DHCP to obtain an IP address. Each packet is directed to the central console's static IP and port.

The desktop console runs a Python application built using the PyQt5 framework. It listens on the configured UDP port for incoming packets, parses the sensor values, and updates a real-time graphical interface. Each sensor has a dedicated display area labeled with units. Users can initiate recording sessions through an interactive dialog that allows sensor selection, time duration, and output file format. Data is saved locally in either .csv or .xlsx format with timestamps and sensor labels for traceability.

All system components operate on independent power supplies. The transmitter is powered by a 3.7V lithium-ion battery regulated to 3.3V, while the receiver draws 5V from a USB-C connection and uses onboard regulation for 3.3V and ±3V analog rails. Mechanical enclosures support stable optical alignment, sensor exposure, and secure wall mounting. The design maintains optical-only data transmission between hubs to meet EMI restrictions, while Ethernet provides reliable, low-latency communication to the user interface.

System Block Diagram

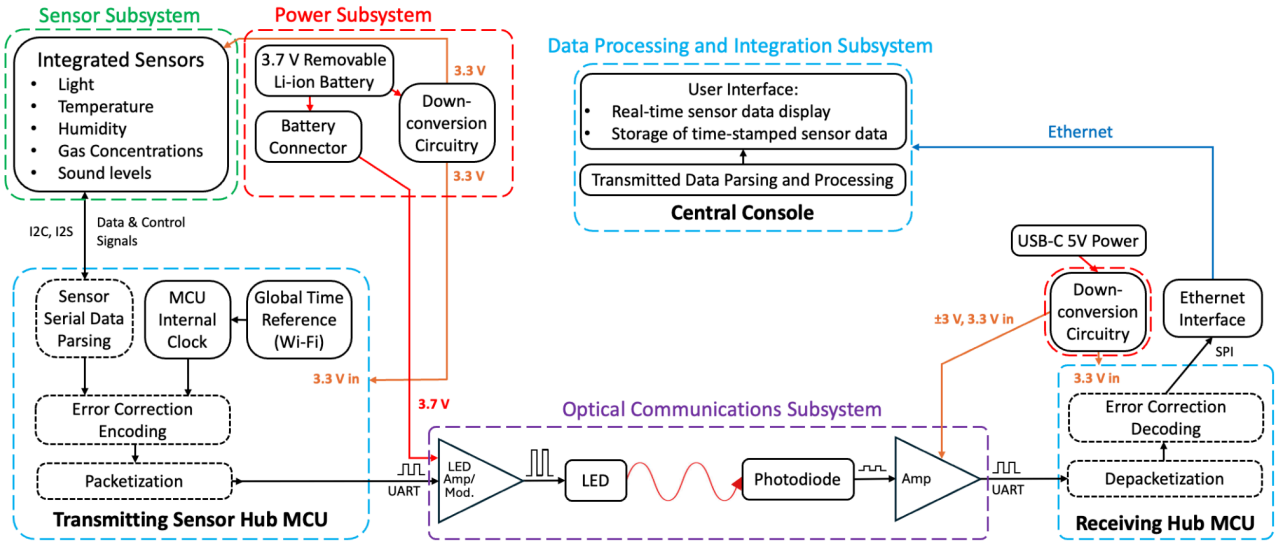

Figure 1. Full System Block Diagram

Detailed Design and Operation of Sensor Subsystem

The sensor subsystem is responsible for monitoring environmental conditions (light, temperature, humidity, gas, and sound levels). This data is critical for the system's overall objective of environmental monitoring. Major Components include:

- SPH0645LM4H-B MEMS Microphone
- BH1750 Light Sensor
- BME680 Environmental Sensor

Sensor Functionality Overview

The following section provides a detailed breakdown of how each sensor operates, including the principles behind their measurements and the reasons for their selection.

A.) BME680

Why chosen:
The BME680 was selected for its ability to provide **temperature**, **humidity**, **barometric pressure**, and **gas** (**VOC**) measurements in a single compact package. This integration minimizes component count, reduces power consumption, and optimizes space, making it ideal for embedded and portable systems.

Temperature

- **Range:** –40 °C to +85 °C
- **Accuracy:** ±1.0 °C

How it works
Uses an integrated band-gap temperature sensor or thermistor on the chip. It detects changes in voltage across a semiconductor junction as temperature varies.

Purpose & Use Cases

- Provides accurate environmental temperature readings.
- Crucial for compensating the humidity and gas measurements.

Humidity

- **Range:** 0% to 100% RH
- **Accuracy:** ±3% RH

How it works
Based on a capacitive sensing element with a hygroscopic polymer that absorbs moisture. As the

water content changes, the dielectric constant shifts, altering capacitance, which is then measured.

Purpose & Use Cases

- Detects relative humidity levels in the air.
- Important for maintaining indoor comfort, preventing mold, and controlling ventilation systems.

Barometric Pressure

- **Range:** 300 to 1100 hPa
- **Accuracy:** ±1 hPa

How it works
Employs a piezo-resistive pressure sensor with a flexible membrane. As pressure changes, the membrane deforms slightly, altering the resistance of internal elements. This change is measured and converted to pressure.

Purpose & Use Cases

- In sensitive environments like hospitals, maintaining stable barometric pressure is crucial in areas such as clean rooms, isolation wards, and surgical suites, where pressure differentials help prevent contamination or the spread of airborne pathogens.

Gas (Air Quality – VOCs)

- **Output:** Gas resistance in ohms
- **Pollutants detected:** Alcohols, carbon monoxide, and general Volatile Organic Compounds from paint, cleaning products, etc.

How it works
Incorporates a Metal-Oxide (MOX) gas sensor, typically tin dioxide (SnO_2), which is heated. VOCs in the air react with the heated surface, changing its resistance based on gas concentration.

Advanced Use
Raw gas resistance data can be processed by Bosch's BSEC software to generate an Indoor Air Quality (IAQ) index.

Purpose & Use Cases

- Tracks indoor air quality for health and ventilation optimization.
- Enables smart control systems to react to pollution spikes or poor air.

Programming Language

- C, chosen because PlatformIO supports all necessary Arduino-compatible libraries, making development straightforward and well-supported

B.) BH1750

Why chosen
The BH1750 was selected for its high precision in measuring illuminance, its digital I²C interface, and its ability to output light intensity directly in lux without requiring manual calibration or conversion. Its spectral response is tuned to the human eye, making it ideal for smart environments and power-aware IoT applications.

Light Intensity

- Measurement Type: Illuminance (visible light intensity)
- Output Unit: lux (lumens per square meter)
- Interface: I²C digital output — no manual conversion needed

How it works
The BH1750 contains a photodiode that generates a current proportional to the amount of visible light it receives. An onboard ADC (analog-to-digital converter) then converts this current into a digital signal. The sensor directly outputs the measured lux value over the I²C bus, simplifying integration into microcontroller-based systems.

Spectral Response

- Covers the 400–700 nm visible spectrum (aligned with human eye sensitivity)
- Minimally affected by infrared and ultraviolet light, improving real-world accuracy in natural and artificial lighting conditions

Purpose & Use Cases

- Auto-brightness adjustment for screens and displays
- Smart lighting that adapts based on ambient conditions
- Daylight detection for indoor/outdoor IoT systems
- Light-triggered power management, e.g., putting devices to sleep when the environment gets dark
- In sensitive environments like hospitals, labs, clean rooms, and care facilities, the BH1750 ensures precise ambient lighting for:
    - Patient recovery and circadian rhythm support
    - Preventing light-induced stress in ICUs and neonatal wards
    - Maintaining optimal visibility in surgical or technical areas

- Reducing contamination risks by minimizing manual lighting adjustments in sterile settings

Programming Language

- C, chosen because PlatformIO supports all necessary Arduino-compatible libraries, making development straightforward and well-supported

C.) SPH0645LM4H-B – MEMS Digital Microphone

Why chosen
The SPH0645LM4H-B was selected for its compact size, low power consumption, and digital I²S interface, which allows for high-fidelity sound capture without analog signal degradation. Its direct-to-digital output simplifies integration and minimizes noise — critical in environments where clean, accurate acoustic monitoring is essential.

Sound (Acoustic Monitoring)

- **Measurement Type:** Sound pressure level (audio intensity)
- **Output Format:** Digital PDM via I²S
- **Interface:** I²S — avoids analog noise and simplifies MCU integration
- **Bit Depth:** 24-bit PCM (via I²S conversion)
- **Frequency Response:** ~100 Hz to 10 kHz
- **SNR:** ~65 dB
- **Dynamic Range:** ~100 dB

How it works
The microphone uses a MEMS (Micro-Electro-Mechanical System) diaphragm that vibrates in response to sound waves. These vibrations are converted to an electrical signal by a capacitive sensor, then digitized internally before being output over the I²S bus. This digital path ensures cleaner signals and better immunity to EMI (electromagnetic interference).

Purpose & Use Cases

- Ambient sound monitoring for detecting anomalies (e.g. sudden noise spikes, alarms)
- Machine health tracking via audio signatures
- Noise-level monitoring in shared spaces
- Smart environmental response (e.g. dimming lights or sending alerts based on sound cues)

In sensitive environments like **hospitals, clean labs, or assisted living facilities**, the microphone enables:

- Non-intrusive acoustic surveillance in ICUs, neonatal wards, or operating rooms
- Real-time detection of equipment beeps, human distress sounds, or unexpected noise events
- Maintaining quiet zones to reduce patient stress and ensure compliance with acoustic safety standards

Programming Language

- C, chosen because PlatformIO supports all necessary Arduino-compatible libraries, making development straightforward and well-supported

Simple Process Flowchart

The following flowchart illustrates the software logic used to initialize and read environmental data from the sensors, highlighting key decision points and processes in the data acquisition cycle.

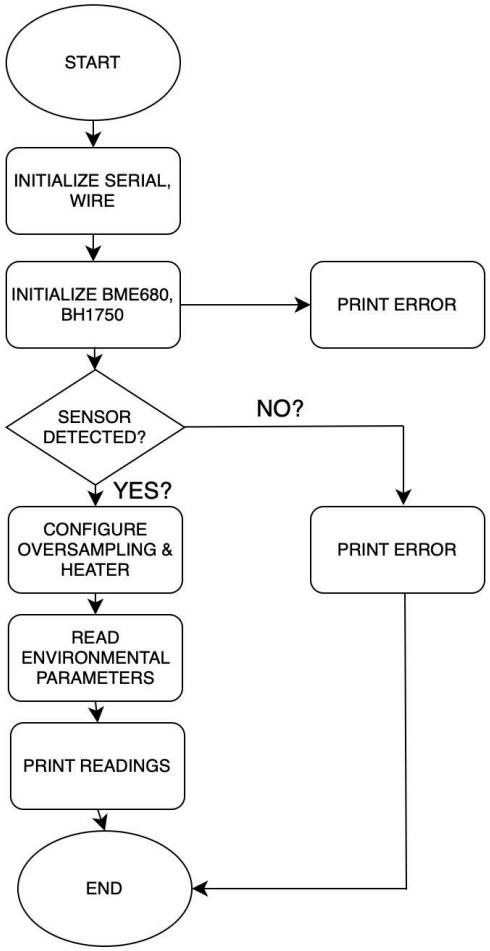

Figure 2. Sensor Operation

The software is loop-driven and follows a linear polling structure, where the microcontroller continuously checks the BME680, BH1750 and MEMs Microphone sensors for new data in a repeating cycle. It is not interrupt-driven and does not use state diagrams. Instead, the program sequentially initializes the sensor, configures its parameters, performs a reading, and prints the data to the serial monitor, repeating this process at fixed intervals using `delay()`. This approach ensures simplicity and reliability in collecting periodic environmental measurements.

The figures below show some snippets of the sensor firmware I generated to read environmental parameters.

```
// ==== MIC READ ====
void MIC_Read() {
    int32_t sample_raw;
    size_t bytes_read;

    i2s_read(I2S_NUM_0, &sample_raw, sizeof(sample_raw), &bytes_read, portMAX_DELAY);

    if (bytes_read > 0) {
        int32_t sample = sample_raw >> 8;

        // Sign extend
        if (sample & 0x00800000) {
            sample |= 0xFF000000;
        }

        int32_t magnitude = abs(sample);
        if (magnitude < 1) magnitude = 1;

        float dB_SPL = SPL_REF + 20.0 * log10((float)magnitude / MAG_REF);

        Serial.printf("Sound Level: %.2f dB SPL\n", dB_SPL);
    }
}
```

Figure 3. Microphone Firmware

```
void BME_Measure()
{
  Serial.println("\nReading BME680 sensor data...");

    // Perform measurement
    if (!bme.performReading()) {
        Serial.println("Failed to perform reading :(");
        return;
    }

    // Print sensor values
    Serial.print("Temperature: ");
    Serial.print(bme.temperature);
    Serial.println(" °C");

    Serial.print("Humidity: ");
    Serial.print(bme.humidity);
    Serial.println(" %");

    Serial.print("Pressure: ");
    Serial.print(bme.pressure / 100.0);  // Convert to hPa
    Serial.println(" hPa");

    Serial.print("Gas Resistance: ");
    Serial.print(bme.gas_resistance / 1000.0);  // Convert to kΩ
    Serial.println(" kΩ");
}
void BH_Measure()
{
  float lux = lightMeter.readLightLevel();  // Read light level in lux

    Serial.print("Light Intensity: ");
    Serial.print(lux);
    Serial.println(" lux");

    delay(2000);  // Wait 1 second before next reading
}
```

Figure 4. Sensor Firmware

Detailed Design and Operation of Power Subsystem

The power subsystem is designed to deliver stable and reliable power to all critical components in both the transmitting hub and receiving hub, supporting the operation of microcontrollers, sensors, optimal communication circuitry, and data interfaces. The system must operate efficiently on both battery and USB-based or Power over Ethernet (PoE) connections. Key requirements for this subsystem include:

- Efficient voltage regulation from a 3.7V lithium-ion Battery to 3.7V - 3.3V for the transmitter hub
- USB-based power or Power over Ethernet (PoE) for the receiver Hub
- Provides sufficient current for all operating components (peak load ~ 600mA)
- Subsystem compatibility with both high-speed digital and analog communication

Interfaces to other Subsystems

- Sensor Subsystem
    - Interface type: 3.3V rail,. I2C
    - Direction: Output/Data
    - Description: Supplies power and receives battery status
- Optical communication subsystem
    - Interface type: 5V rail,. GPIO
    - Direction: Output
    - Description: Powers LED and receiver circuitry
- Data processing subsystem
    - Interface type: 3.3V rail,. SPI
    - Direction: Output
    - Description: Powers Ethernet module and MCU

A common ground plane is maintained across all subsystems to ensure voltage stability and signal noise.

Circuit Description and Schematics:

- Transmitter Hub
    - Power Source: 3.7V 2500mAh Lithium-ion (removable) Battery
    - Voltage Regulation: Buck-boost converter (TPS63802), steps 3.7 to 3.3V
    - Voltage Divider: 100kΩ and 68kΩ resistors to scale voltage to safe ADC range
    - Powered Components: ESP32-S2, BME680, BH1750, microphone, IR LED
- Receiver Hub
    - Power Source: 5V USB via PoE
    - Voltage Regulation: LDO Regulator steps down 5V to 3.3V
    - Photodiode Circuit: Uses LM27762 charge pump to generate ± 3V
    - Powered Components: ESP32-S3, W5500 Ethernet, photodiode receiver

Schematics for both the transmitter and receiver hub power paths are attached in ***Appendix A***. These show regulator placement, decoupling capacitors, etc.

Component Selections and Justification:

- **TPS63802 (Buck-Boost Converter):** Allows power delivery even when battery voltage drops below 3.3V, unlike LDOs. High efficiency (85–90%) minimizes heat and extends battery life.

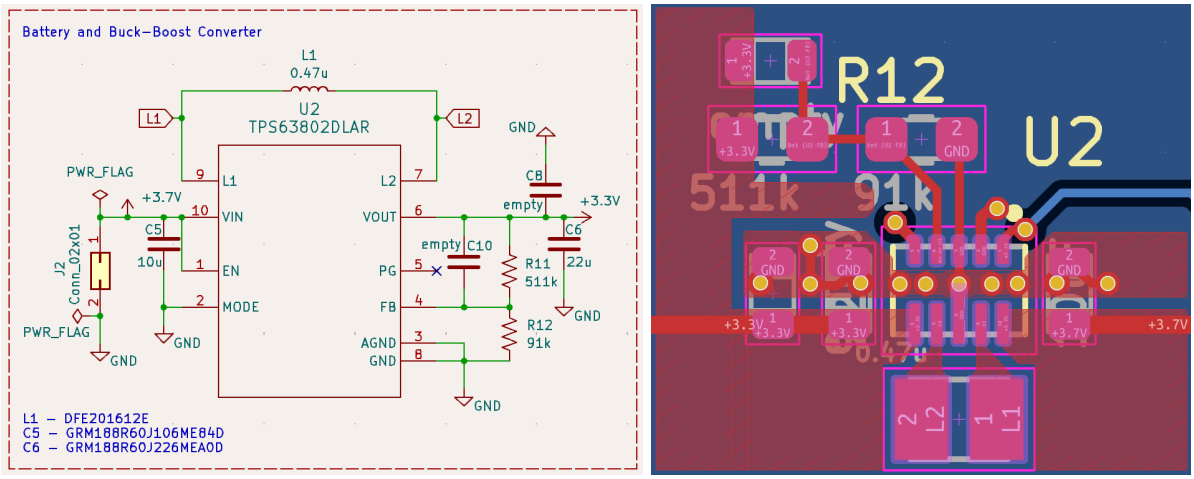

Figure 5. 3.3V DC-DC Buck-Boost Converter

- **ESP32-S2/S3:** Low-power MCUs with integrated Wi-Fi/Bluetooth, reducing external components. Deep-sleep current is as low as 10µA.

- **LDO Regulator for Receiver:** Chosen due to stable 5V input; simplicity favored over switching efficiency.

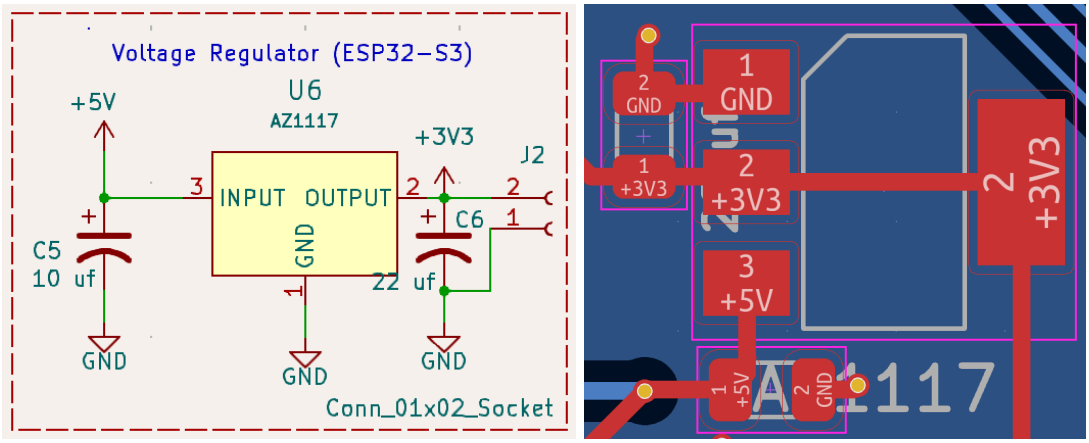

Figure 6. 3.3V LDO Regulator

- **Ferrite Beads & Decoupling Capacitors**: Minimize high-frequency noise and voltage dips near ICs.

Battery Life (without Power optimization) :

To guide our battery selection, we first estimated the system's current draw under typical active conditions and calculated the expected battery life. This analysis ensured that our chosen battery capacity would support at least a full day of continuous operation, even without implementing power-saving features.

Table 1. CURRENT DRAW Power Requirements Summary

Component	Voltage (V)	Current (Active, mA)	Current (Sleep, µA)
ESP32 Microcontroller (modem-sleep mode): **Sensor Board: ESP32-S2** **Receiver Hub: ESP32-S3**	3.3V	**Sensor Board: 24mA** at 160 MHz, CPU and all peripherals on **Receiver Hub:** N/A, powered by USB	**Sensor Board: 10 µA** **Receiver Hub:** N/A, powered by USB
BME680 Sensor (Temp, Humidity, Pressure, Gas)	1.71V - 3.6V	0.09 - 12 mA → **12 mA**	**0.15 µA**
BH1750 Sensor (Light)	2.4V - 3.6V	0.12 - 0.18 mA → **0.12mA**	**0.01µA**
I²S SPH0645LM4H Microphone (Audio)	1.62V - 3.6V	600 µA	**10µA**
Infrared LED and associated circuitry **Sensor Board: VSLY5940**	3.7 V	**Sensor Board: 31.25 mA** **Receiver Hub:** N/A, powered by USB	N/A, LED either on or off NMOS leakage: **10 µA**
BPW34 Photodiode and associated circuitry	5 V or 3.7V (amplification) and 3.3 V (comparator)	**Sensor Board:** N/A, no optical receiver **Receiver Hub:** N/A, powered by USB	Comparator always on, negligible IC turn-on times. **Receiver Hub:** N/A, powered by USB
TPS63802 DC-DC Converter Efficiency (Sensor Board)	85% - 90%	N/A	N/A
W5500 Ethernet Module (Receiver Hub)	3.3V	N/A, powered by USB	N/A, powered by USB
Total (Sensor Board)		**67.37mA + 620µA= 67.37mA + 0.620 mA = = 67.99 mA**	**20.16µA = 0.02016 mA**

Estimate battery life for continuous operation at an average current draw of 67.99mA
Battery type: Lithium-Ion Polymer Battery of 3.7 V / 2,500mAh - [3.7v 2,500mAh]

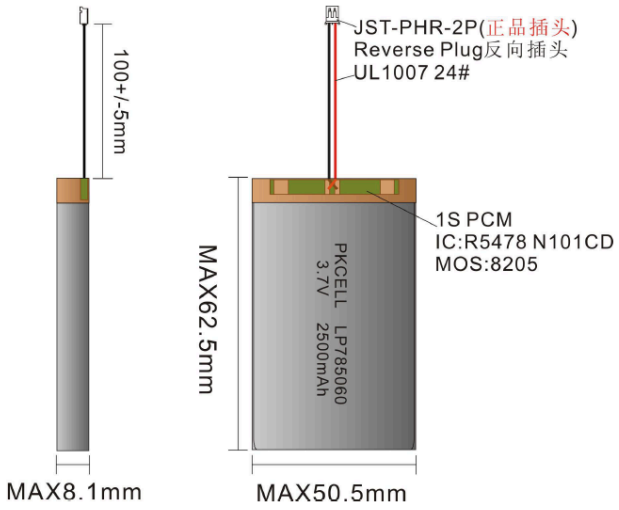

Figure 7. Adafruit: Lithium-Ion Polymer Battery - 3.7v 2500mAh

- Energy available from the battery:
  - Battery specification:
  - Capacity = 2,500 mAh (1.5Ah)
  - Voltage = 3.7V
  - Total energy stored : $E = V \times C$
  - $3.7V \times 2.5Ah = 9.25\,Wh$
  - The battery can deliver 9.25 watt - hours of energy before depletion

- Power Consumption of System:
  - System operates at 3.3V so power usage:
  - Current draw : 68mA = 0.068A
  - Power consumption: $P = V \times I = 3.3V \times 0.068A = 0.224W$
  - Meaning that the system consumes 1.14 watts continuously

- Runtime without optimization:
  - $runtime = \frac{battery\ energy}{power\ consumption}$
  - $runtime = \frac{9.25\,Wh}{0.224W} = 41.29hrs$
  - Without any optimizations, the 2,500mAh battery would last about 41.29 hours or 1.72 days before depletion

While current measurements reflect continuous operation, the outlined estimates shown in *To-Market: Design Changes* section, highlight the value of further optimizing power consumption to extend operational longevity.

Detailed Design and Operation of Optical Communication Subsystem

The optical communication subsystem will be responsible for reliable optical transmission and reception of data and control signals between the transmitting sensor devices and the receiving hub devices. Key requirements include:

- Successful transmission and reception of data, i.e., the received communication signal shall overcome noise/interference, both optical (reduced via filters and wavelength selection) and electrical (reduced with low-noise electrical components and design).
- Transmission rate: $\geq\ 100\ kbps$
- Transmission range: $\geq\ 3\ meters$
- Data encoding shall be applied to to support correction of errors introduced by optical transmission
- The optical communication system should aim for lower power consumption to relax the battery requirements.

Signal Format and Circuitry Bandwidth

To satisfy the speed requirement, we consider 2 optical signal formats:

1. Standard on-off keying (OOK) at 115.2 kbps

In this format, each symbol is a pulse of length $\frac{1}{115.2\ kbps}$ = 8.7 μs. The corresponding sinc function frequency response has nulls at multiples of 115.2 kHz. To create fast transitions and prevent inter-symbol interference, a standard rule of thumb is to design the corresponding circuitry to include 10 nulls (i.e. design for a bandwidth of 1.152 MHz).

2. IrDA: UART based protocol at 115.2 kbps

In this format, the symbol length is $\frac{1}{115.2\ kbps}$ = 8.7 μs; however the actual bit =1 pulse is $\frac{3}{16} \times\ 8.7\ \mu s$ = 1.63 μs long (see Figure O1). The separation between pulses allows for increased robustness to inter-symbol interference and a relaxed requirement for the number of nulls of the pulse's frequency response required for a distinguishable signal. A standard rule of thumb is to include 2 nulls (i.e. design for a bandwidth of $2\ \times \frac{1}{1.63\ \mu s}$ = 1.22 MHz).

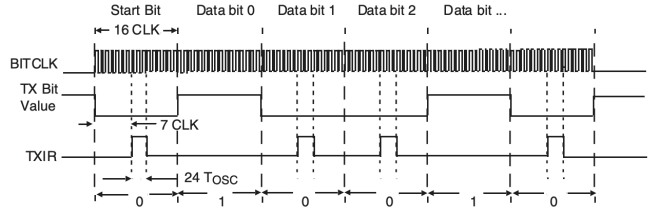

Figure O1. IrDA signal format

To allow for the implementation of both OOK and IrDA, we design our optical communication system for a circuitry bandwidth of 1.22 MHz.

Optical Communication Hardware Design

Optical Components

To avoid visible light noise, we choose infrared optical components. Following the requirements, the optical components must be able to generate and detect 1.63 µs pulses, so the rise and fall times must be significantly shorter. For conversion of fast optical signals to an electrical signal, photodiodes (PDs) are often used. Additionally, the LED transmit power and PD sensitivity must be sufficient to transmit over 3 m.

An LED and PD with these qualities are the VSLY5940 and the BPW34. The relevant properties are:

LED: VSLY5940	**Photodiode: BPW34**
• Wavelength: 940 nm • Intensity: 600 mW/Sr @ 100 mA, intensity vs. forward current characteristic on datasheet • Rise/fall time = 10 ns • IV characteristic on datasheet	Rise/fall time = 100 ns Radiant area = 7.5 mm^2 I vs E_e characteristic on datasheet Capacitance: 25 pF @ V_B = 3 V, 70 pF @ V_B = 0 V

Considering the PD's radiant area, the LED intensity at a driving current of 100 mA, PD's current vs. intensity characteristic, and transmission distance of 3 m, the expected current is 0.33 µA. While small, this current can be amplified to be readable by the ESP32. While a driving current of 100 mA is used in calculations, to account for optical misalignment and part tolerances, we use a 5 Ω resistor in the LED driving circuit, which translates to a driving current of ~300 mA and ~3x the emitted optical power (See Figure O3).

While the BPW34 has peak sensitivity at infrared wavelengths, it is also sensitive to visible light. We use an infrared longpass filter to isolate the response to infrared radiation (see Figure O2).

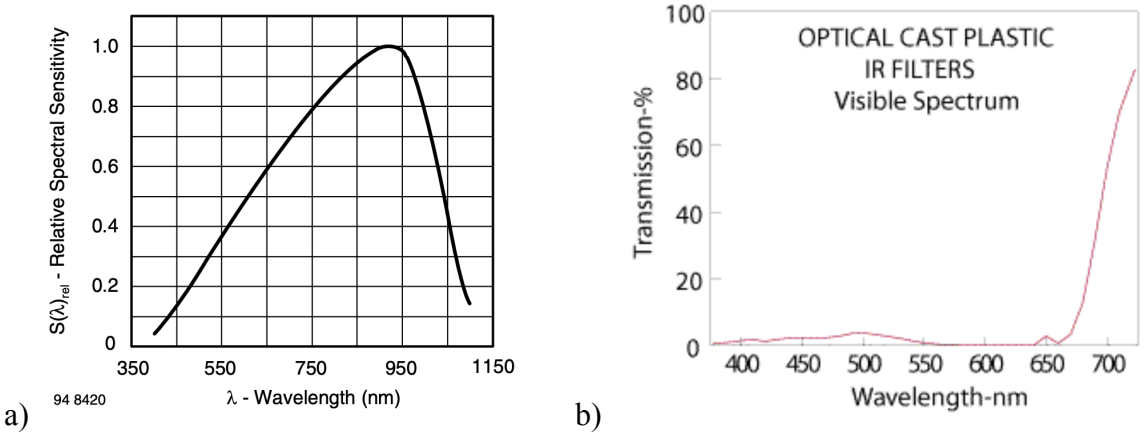

Figure O2. a) BPW34 spectral sensitivity and **b)** IR filter spectral response (see datasheets)

Optical Subsystem Electrical Design

Transmitter

To convert 3.3V digital signals from the MCU GPIO to 300mA pulses on the LED, we use an NMOS as a high-speed, high current switch. We have chosen the DMG2302UK-7 because it sufficiently fast (rise/fall time: < 5ns << 1.63 µs), it works up to 2.4 A (> 1 A), its threshold V_{GS} = 0.6 V is lower than and its maximum input voltage $V_{GS,max}$ = 20 V is higher than the MCU logic voltage (3.3V), and it has a small on-resistance $R_{DS,on}$ = 120 mΩ @ V_{GS} = 2.5 V. Decoupling capacitors are added to ensure stable power to the LED. The schematic is shown below:

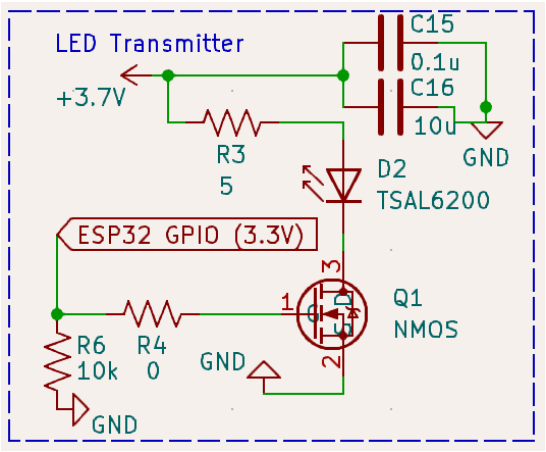

Figure O3. Optical Transmitter Schematic

The transistor is off for a majority of the time. We design the power consumption for OOK transmission due to its higher pulse length and thus higher power consumption. For OOK transmission, the signal is high only 1/2 of the time (assuming an equal number of 0's and 1's in the transmitted data) during transmission. If the transmitter is only transmitting ⅛ of the time; the equivalent continuous current is $100\,mA \times 1/2 \times 1/8 =$ 31.25 mA.

Receiver

The receiver must convert 0.33 µA signals from the photodiode to 3.3 V digital signals readable by an ESP32 GPIO pin. A transimpedance amplifier (TIA) serves as an initial amplification stage, a unity-gain inverting amplifier inverts the TIA stage output to the desired polarity, and a comparator converts the small TIA output voltage into a 3.3 V digital signal. To reiterate, the bandwidth requirement is 1.22 MHz and low noise electrical components and power supplies should be used.

A comparator that satisfies these requirements is the TLV3501. The TLV3501 has a maximum toggle frequency of 80 MHz (>> 1.22 MHz), outputs its supply voltage (which can be provided by the 3.3V rail to interface with ESP32 GPIO pins), and has a minimum input voltage of 0.3 V (which determines the TIA gain required). Additionally, the comparator requires power between 2.7 V and 5.5V and has a quiescent current of 3 mA.

An op-amp that is designed for high gain, high BW, low noise amplification is the OPA657. The OPA657 has a high gain-bandwidth product (GBP) = 1.6 GHz. To reach 0.3 V output for the comparator from an input of 0.33 µA, the feedback resistance R_F = 1 MΩ. The bandwidth is calculated as:

$$f_{-3dB} = \sqrt{GBP / (2\pi R_F C_D)}\ Hz$$

For R_F = 1 MΩ and a diode capacitance of 25 pF, f_{3dB} = 3.2 MHz > 1.22 MHz.

An op-amp designed for low-noise, unity-gain, relatively high speed signal inversion is the OPA192. The OPA192 has a unity gain-bandwidth of 10 MHz > 1.22 MHz.

The OPA657 has a quiescent current of 16 mA, and the OPA192 has a quiescent current of 1 mA. Both op-amps operate using a dual supply voltage, which requires a dedicated power supply. While the OPA657 datasheet states that the minimum rated supply voltage is ±4V, tests with a desktop power supply have shown that the OPA657 works as expected with a supply voltage of ±3 V. Looking toward future designs where receiver circuitry is battery powered, the dual supply voltage is designed to be ±3 V. Given 3 V is lower than the typical 3.7 V supplied by a lithium-ion battery, a ±3 V dual supply can be generated from the battery using a low-noise charge pump IC, such as the LM27762. A power supply voltage of ±3 V is within the rated range for the OPA192.

The power dissipated by the feedback resistors is negligible because the input signal currents are extremely low, and the output current of the TIA and inverting amplifier stages is negligible because the input impedances of the subsequent stages are on the orders of 10^{12} Ω.

In order to account for variation in received power due to installation, a manual 200 kΩ potentiometer (3296W-1-204RLF) is included at the inverting input of the comparator to adjust the comparator threshold voltage. The receiver circuitry is shown in Figure O4.

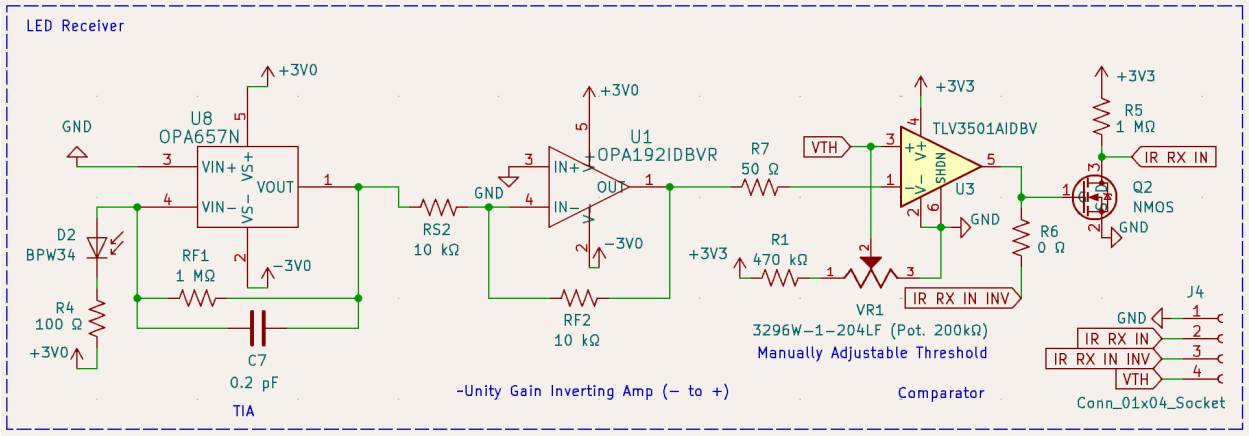

Figure O4. Optical Receiver Schematic

The voltage divider at the output of the comparator dissipates approximately 5 µA of current.

To provide stable ±3 V from 5 V, with a current requirement of 16 mA + 1 mA = 17 mA, the LM27762 (390 uA quiescent current) is used (maximum output current = 250 mA >> 17 mA). Resistor values are chosen to generate ±3 V. Decoupling capacitors are included for a stable supply voltage. The schematic is shown below:

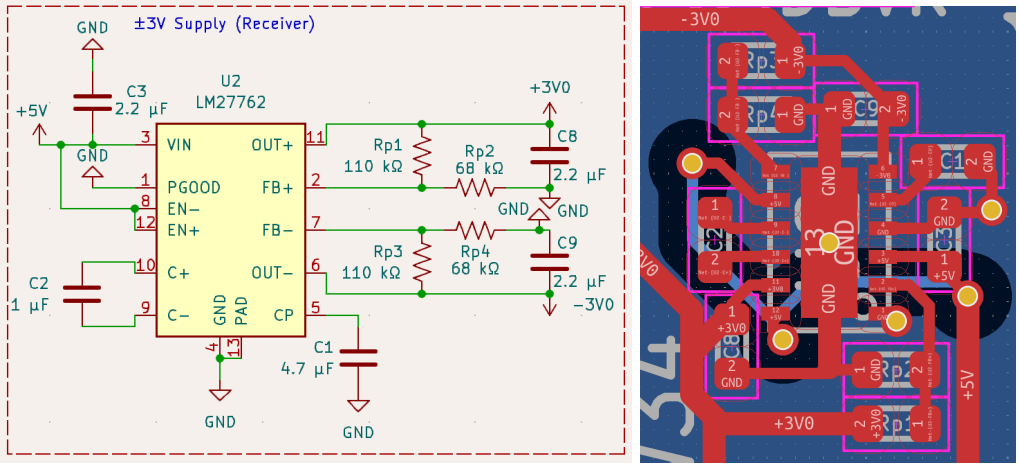

Figure O5. ±3V power supply

Optical Communication Code Structure

The optical communication code forms the critical link between the transmitter and receiver hubs in our project. The code is divided into two primary functional domains: UART-based optical signal transmission and optical signal reception with error correction. For coding simplicity, standard 115.2 kbps UART signals were outputted using standard Arduino-C UART libraries from an ESP32-S3 GPIO pin to modulate the IR intensity of the transmitting LED. To correct for bit errors arising from electrical noise and a noisy and/or slow transition between high and low signal levels, the transmitted data was encoded with Hamming (8,4) codes

Transmitter Code Structure

The transmitter code is responsible for encoding and modulating data to be sent over the optical channel. After initializing the UART peripheral, the system formats the outgoing payload as a character string, representing the encoded data to be transmitted. Prior to UART output, the data undergoes Hamming (8,4) encoding, each 4-bit segment of data is expanded into an 8-bit word containing four parity bits. This encoding increases resilience to single-bit errors introduced during optical transmission.

Encoded bytes are transmitted using the UART hardware peripheral, where each bit is represented by a digital voltage level. These digital signals toggle a GPIO pin connected to the gate of a high-speed NMOS transistor. The transistor acts as a switch for the infrared LED, converting the digital UART waveform into a modulated optical signal. The result is an On-Off

Keying (OOK) optical stream with high pulses for logical '1's and corresponding gaps for '0's. A newline character is appended to each packet to signal the end of a transmission.

The transmitter firmware uses a simple loop structure. It periodically encodes and transmits the data once per second. Timing is controlled using a blocking delay, and the UART transmission is non-blocking, handled by the hardware buffer. No interrupts are used, simplifying timing control and reducing processor overhead.

Receiver Firmware Structure

On the receiver end, a dedicated UART interface listens for incoming pulses from the optical link. The infrared signal is first captured by a photodiode and passed through a transimpedance amplifier, inverter, and comparator. This analog front-end converts low-current optical pulses into a clean digital waveform, which is then fed directly into the UART receive pin on the ESP32-S3.

In firmware, the UART input is buffered into a character array. The program polls the UART stream, accumulating characters until a newline is detected. Once a full packet is received, the data is segmented into 8-bit blocks for decoding. Each block undergoes Hamming (8,4) error correction, checking for single-bit errors and correcting them based on the parity bits embedded in the transmission.

After decoding, the restored data is reconstructed into a clean ASCII string. This output is not processed locally, instead it is passed to a communication module (e.g., SPI-to-Ethernet) for relay to the central console. The core logic remains in the optical decoding loop, which ensures correction of 1-bit errors.

Error Correction Logic

As previously mentioned, Hamming (8,4) codes are implemented. At the transmitter, a Hamming encoder transforms each 4-bit data chunk into an 8-bit encoded byte. The receiver uses a decoding function that checks parity bits and applies a syndrome-based correction to any single-bit errors found in the byte.

This choice of error correction balances efficiency and computational simplicity, aligning well with the processing and timing constraints of the ESP32 microcontroller and the bandwidth limitations of the IR channel.

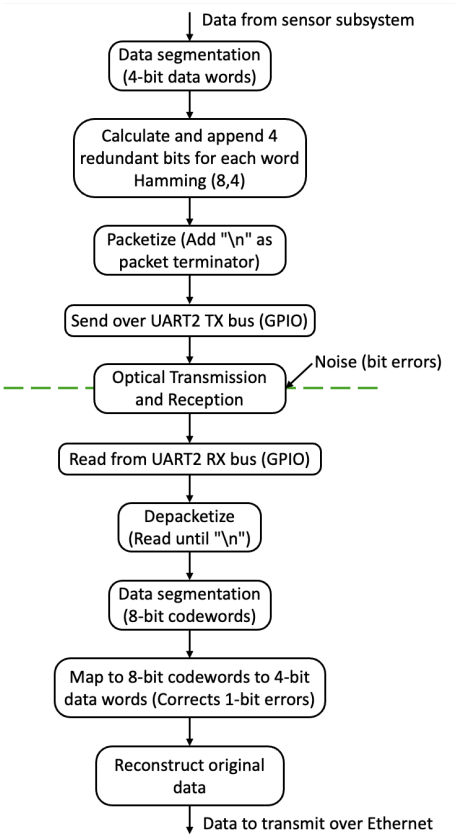

Figure O6. Optical Communication Code Structure

Detailed Design and Operation of Data Processing and Integration Subsystem

The Data Processing and Integration Subsystem is responsible for transferring decoded sensor data from the receiver hardware to the central console for visualization and recording. This subsystem integrates low-level Ethernet communication, UDP data transmission, and a Python-based graphical user interface (GUI), enabling seamless data flow across the network.

To support Ethernet-based transmission, the system uses the W5500 Ethernet controller. This chip was selected for its widespread availability in breakout board form, simplifying initial prototyping. Following successful validation during bench testing, the W5500 was integrated into the final design to reduce development time and maintain continuity across hardware revisions.

The single receiver board includes a W5500 chip whose MAC address is explicitly defined in firmware using the byte mac[] declaration. This fixed hardware address provides a consistent identity for the device during network configuration and communication. Dynamic IP assignment is handled using DHCP, initiated with Ethernet.begin(mac), allowing the W5500 to request a valid IP address on the university subnet. The IP address of the desktop console is hardcoded in the firmware using IPAddress desktopIP(...) to provide a reliable destination for packet delivery. A static UDP port (4210) is defined and matched in both the embedded firmware and the GUI application to maintain a consistent communication pathway.

The transmission protocol used between the microcontroller and the central console is UDP (User Datagram Protocol). UDP was selected for its lightweight nature and suitability for real-time applications where speed is prioritized over guaranteed delivery. The firmware uses the Udp.beginPacket() and Udp.endPacket() methods to send decoded sensor values, formatted as a CSV string, to the desktop GUI once per second.

The ESP32-S3 microcontroller communicates with the W5500 via SPI. All necessary SPI lines, including chip select and reset, are defined and initialized in code. The reset line is pulled low and then high on startup to reinitialize the Ethernet controller. Sensor data decoded from the infrared optical subsystem is received through a dedicated UART interface and relayed through the Ethernet module.

Deployment on the university network required coordination with the Office of Information Technologies (OIT). The MAC address of the W5500 was registered to a hostname to allow the device to obtain DHCP leases. Additionally, ND Network Services activated UDP port 4210 to allow cross-subnet delivery of sensor packets to the console machine. These permissions were essential for establishing functional communication and complying with university network policies.

The receiver software follows a polling model in its main loop. Incoming serial data from the infrared decoder is buffered and processed on demand; no interrupts are used. When valid data is received, it is transmitted immediately over UDP. The desktop GUI, developed in Python using PyQt5, binds to the same UDP port and processes incoming messages using a

non-blocking socket. Sensor values are parsed and rendered in a dedicated display window. Additional functionality includes time-based data logging and export to Excel or CSV.

Testing of this subsystem was performed at both hardware and software levels. An initial version of the receiver firmware generated random sensor values, allowing Ethernet communication and GUI display to be tested without requiring the optical link. Real-time status messages printed to the serial monitor confirmed Ethernet readiness, IP assignment, and packet delivery. On the GUI side, incoming packets were monitored, decoded, and checked for consistency. Logging functionality was verified using synthetic datasets, and the complete end-to-end system was validated after integration with the photodiode input.

Integration with Other Subsystems

The Ethernet-based data transmission process represents the final stage of the complete sensing pipeline. This subsystem receives its input from the Optical Communication Subsystem, which transmits Hamming-encoded sensor data over an infrared link from the transmitter board. The receiver board's photodiode module converts these optical pulses to electrical signals, which are decoded and passed to the Ethernet subsystem via UART. As such, the Ethernet transmission acts as a transport layer between the Optical Subsystem and the GUI on the Central Console.

The original sensor readings originate from the Sensor Subsystem, which includes temperature, humidity, pressure, light, gas, and sound sensors mounted on the transmitter board. The Ethernet subsystem does not interact directly with the sensors but transmits their processed values once received through the optical decoder.

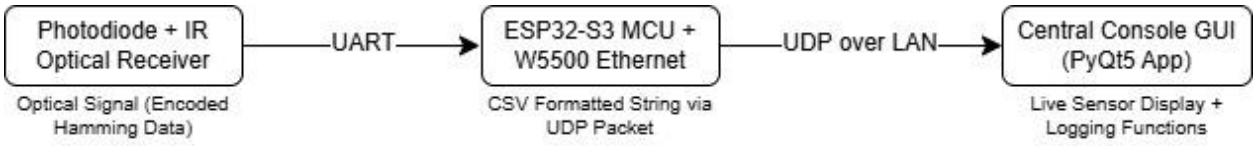

Figure 8. UDP Communication Flow Diagram

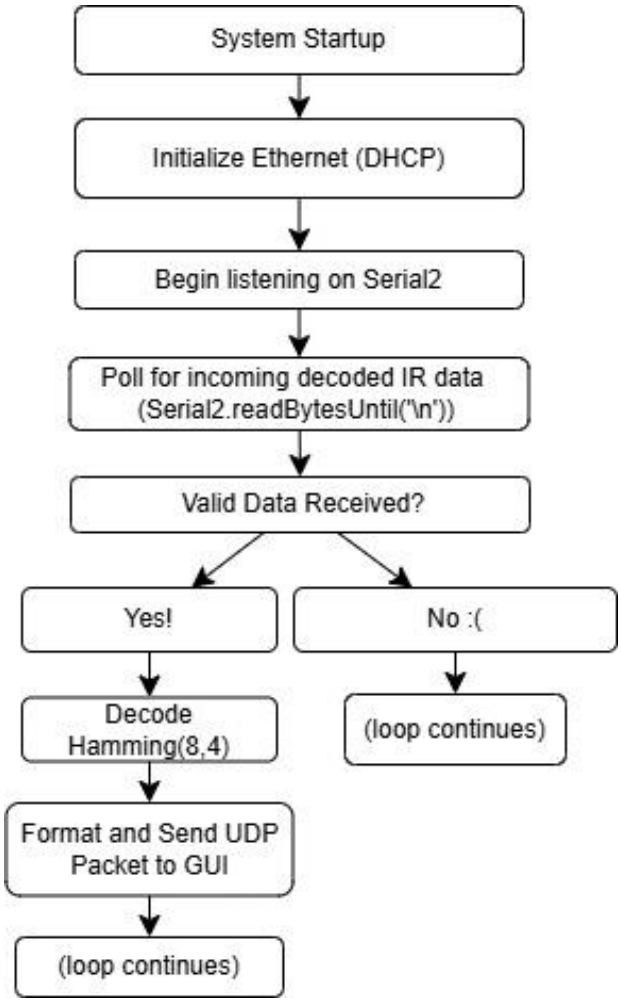

Figure 9. Receiver Main Loop State Diagram

System Integration Testing

System integration testing confirmed that the subsystems for sensing, optical transmission, infrared decoding, Ethernet communication, and user display functioned as a coordinated unit. Testing addressed both system-level functionality and the specific performance targets outlined in the design requirements.

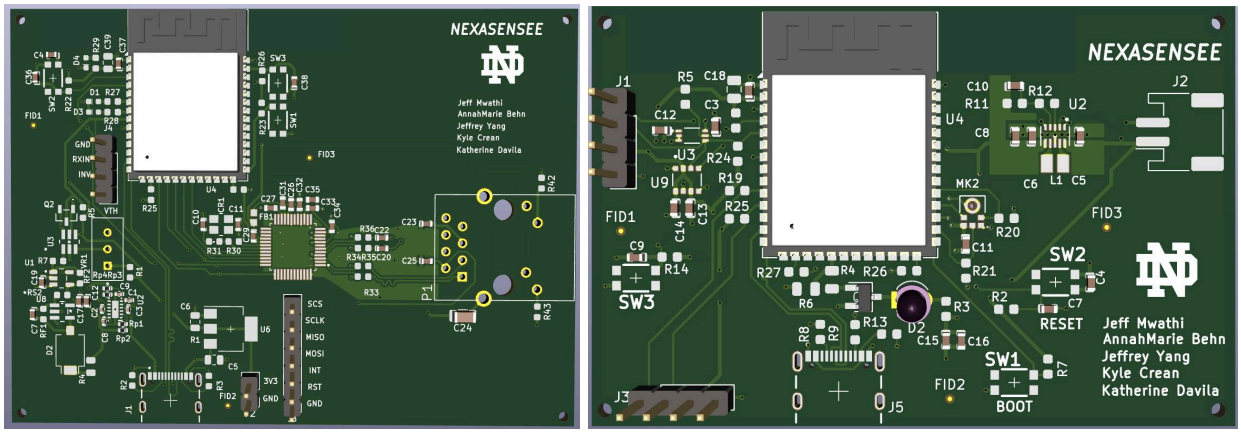

Figure 10. Transmitter and Receiver 3D Printed Circuit Board Models

Test Conditions and Procedure

The full prototype was evaluated in an indoor environment configured to represent a space where electromagnetic emissions must be avoided. The transmitter and receiver were aligned at a fixed distance of 2.5 meters with unobstructed line-of-sight. Initial tests used synthetic data generated by the receiver firmware. This allowed Ethernet and GUI behavior to be assessed before activating the optical data link. Once confirmed, testing proceeded using live sensor values transmitted through the infrared channel.

Functionality and Communication

Sensor data for temperature, humidity, pressure, gas concentration, light, and sound was collected by the transmitter and sent across the infrared link. The receiver interpreted these signals and passed decoded values to the Ethernet interface. Each reading appeared in the corresponding section of the graphical user interface. The GUI updated once per second and provided accurate, labeled data. The recorded output saved in both .csv and .xlsx formats matched the expected time structure and data layout

All transmitted packets were received on the correct UDP port. Packet contents matched decoded sensor outputs. The receiver code printed network status to the serial terminal, confirming that the W5500 Ethernet chip had been assigned an IP address and that link detection was successful. The central console GUI bound to the same UDP port and continuously received data without interruption or delay.

Power and Stability

The transmitter operated within the projected current draw of 150 mA during continuous sampling and optical transmission. The receiver was powered over USB-C and supplied a stable 3.3V rail to both the ESP32-S3 and W5500 through onboard regulators. No thermal issues or voltage drops were observed during extended testing.

Optical Alignment and Housing

The enclosures allowed for direct exposure of the infrared LED and photodiode. This maintained reliable communication across the full operating range. Ambient light conditions were varied to confirm the photodiode amplifier circuit maintained signal fidelity. Housing slots also allowed ambient air to reach the sensors, which responded accurately to controlled changes in temperature and humidity. Mounting the enclosures to vertical surfaces did not introduce alignment problems or signal dropout.

Table 2. Compliance with Design Requirements

Requirement	Outcome
No RF emissions	Verified with passive IR-only communication
Transmission range of a minimum of 3 meters	Confirmed at 2.5 meters with margin
One-second latency	Measured response time: approximately 750 milliseconds
Sensor accuracy	Confirmed with reference data on average room characteristics
150 mA current draw	The transmitter measured between 140 and 145 mA during use
GUI logging and display	Live updates and the correct file export format are validated
DHCP and network access	IP assignment and port access confirmed through coordination with OIT

User Manual

Installation and Setup

To install your NEXASENSEE system, first determine the indoor location you seek to monitor. This location must have at least one set of parallel walls no more than 10 meters (33 feet) apart from each other. There must be at least two Ethernet ports in this room: one for sending data from the receiver hub, and one for loading data to your determined central console.

Receiver Setup

Before installing, tune the potentiometer to an appropriate voltage threshold $V_{TH,}$ so that the comparator can accurately decide if the amplifier output corresponds to an optical pulse. $V_{TH} > 0.8$ V is recommended to avoid reading power supply noise as an optical pulse. If needed, replace R_1 (the other resistor in the divider) with a different value to achieve a different threshold voltage.

For alignment purposes, it is recommended to begin with installing the receiver module. To do so, place the provided Command Strip sticky backing on the module's housing. Next, on a wall with an established Ethernet port, secure the module (Command Strip side facing the wall) at least 7 feet above the ground. Through the Ethernet port opening on the housing, connect the module via Ethernet cord to the Ethernet port on the wall as shown in Figure 1c. Connect the module to power using the provided USB-C cable and wall brick through the USB-C opening. The complete setup of the receiver module can be seen below in Figure 1.

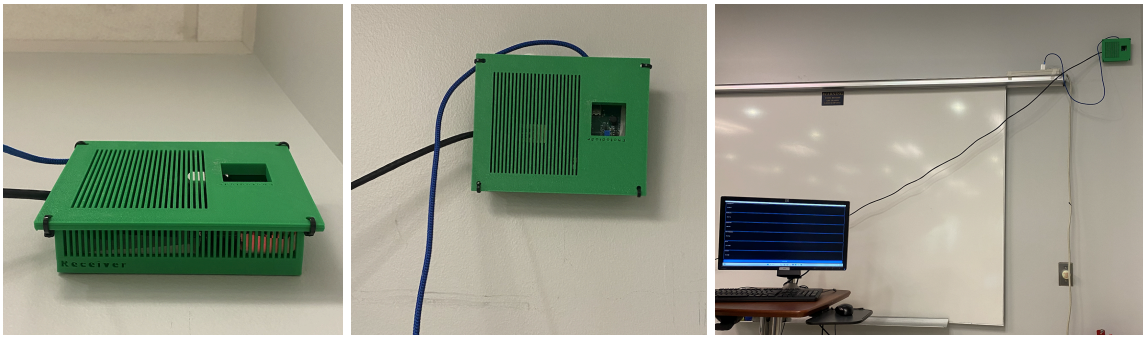

Figure 11. Installation of the Receiver Module on the Wall

Transmitter Setup

Place the provided Command Strip sticky backing on the transmitter module's housing, just as completed in the previous setup procedure. On the wall opposite the receiver module, mount the transmitter module. This installation can be seen below in Figure 2.

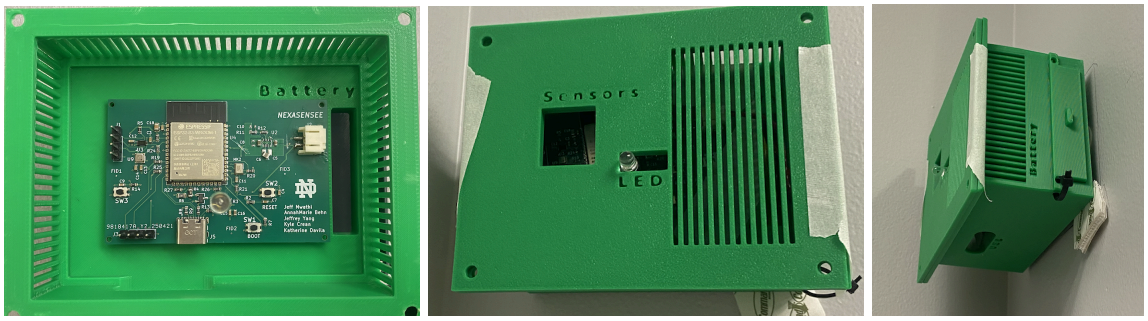

Figure 12. Installation of the Transmitter Module on the Wall

Figure 13. Example Setup of Transmitter and Receiver Modules

Whenever the battery is plugged in, connect to the ESP32 and open the serial monitor to ensure the sensors are working and to begin optical transmission. To ensure a satisfactory optical link during installation, upload the LED aiming programs to the transmitter (tx_aim_LED) and the receiver (rx_aim_LED) via the respective USB-C ports. The transmitter LED aiming program will toggle the transmitter's IR LED every 1 second. The receiver LED aiming program will turn on a visible light LED on the receiver board when the transmitter's LED is on and the alignment is satisfactory. LED leads can be bent if needed for better alignment. After alignment, the user can upload the respective programs to the transmitter and receiver over USB-C. The user can then open the NEXASENSEE_GUI.exe application on their central console to view live data and begin recording if desired.

Figure 14. Full Setup Example

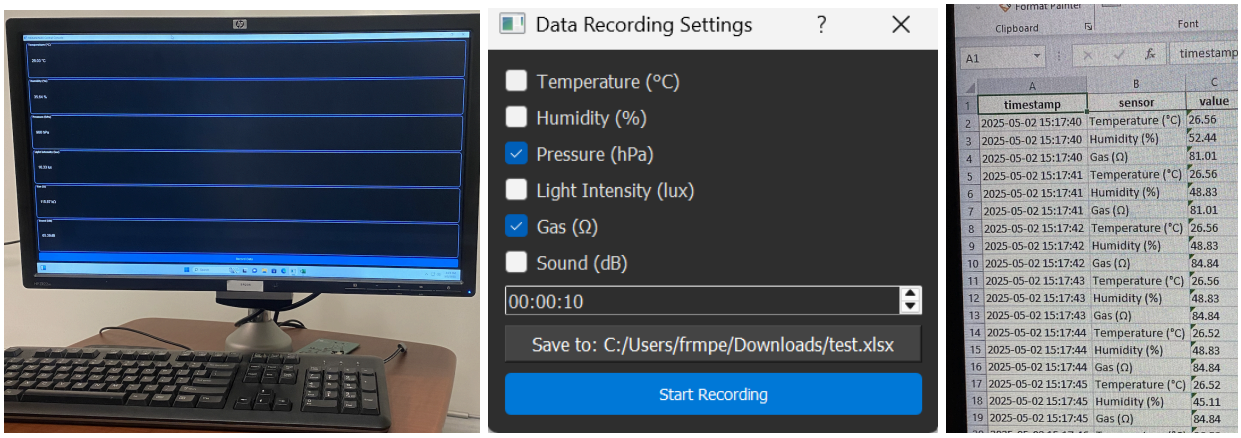

Figure 15. GUI on Central Console, Data Recording Window, and Example Save File

Installation Considerations

- For rooms that experience heavy foot traffic, the ceiling height must be a minimum of 7.5 feet to avoid physical interference with the optical communication path.
- Using a desktop computer as a central console is ideal, as it will be stationary and not require the transfer of Ethernet port connections.

How to Confirm the System is Functional

By running the NEXASENSE_GUI.exe application, the user will observe real-time changes in environmental characteristics updated every two seconds. To confirm general functionality, an initial reference humidity, temperature, gas pressure, sound, and light intensity level should be known to compare the values displayed on the GUI with.

Troubleshooting

If you are not receiving data to the GUI:

1. Confirm your devices are powered
2. Run the official receiver code on the receiver PCB while the receiver is still connected to your programming device
    a. If garbled data appears on your device's serial monitor, the issue is with transmitter - receiver module alignment
3. Run the aiming code
    a. The receiver LED aiming program will turn on a visible light LED once alignment is satisfactory
4. Run the randomly generated data code on the receiver PCB to determine if it is an Ethernet Communication issue
    a. Verify that the W5500 MAC address is explicitly defined in the receiver code and not shared with another device on the network
    b. Verify that the IP address defined in the receiver code matches that of the central console the GUI is displayed on
    c. On the serial monitor, you should see a confirmation message that the Ethernet Cable is detected, and shortly after, a string of values that represent the data being sent over Ethernet.

To - Market Design Changes

Before transitioning our prototype into a market-ready product, several key modifications will be necessary to improve reliability, manufacturability, cost-efficiency, and user experience. These design refinements aim to address limitations discovered during prototyping, enhance long-term performance, and ensure compliance with commercial standards. The following subsections outline the critical changes we intend to implement prior to final production and distribution.

Battery Efficiency:

Future improvements to the power subsystem will focus on implementing deep sleep functionality to significantly reduce power consumption. Although initial attempts were made to integrate deep sleep modes; supported by both the microcontroller hardware and theoretical

runtime projections, firmware-level challenges ultimately prevented successful deployment in the current prototype.

To maximize the efficiency of our energy budget, future efforts will prioritize resolving these firmware issues, enabling the system to enter low-power sleep states between active cycles. By leveraging deep sleep, the microcontroller can drastically reduce its current draw; waking only at predefined intervals to perform sensing and transmission tasks. This change would transform the current continuous draw model into a duty-cycled one, vastly improving energy efficiency.

Estimates, as shown below, suggest that implementing deep sleep could extend battery life from under two days to several weeks, depending on the sensing interval and active time. These improvements are especially important for remote or long-term deployment, where recharging or replacing batteries is impractical.

Battery life calculations with power optimization

Estimate of how long the battery will last under power optimization and comparing different scenarios of frequency of data collection (determining when in active mode and when in sleep mode).

The total current draw of the system is a combination of:

1. Active Mode: system wakes up, collects data, transmits, then goes back to sleep.
    a. Draws ~ 68 mA for a short duration (2 seconds per cycle).
2. Deep-Sleep Mode: system is in low power mode.
    a. Draws ~ 0.02016 mA when sleeping.
3. Duty Cycle Impact: The more frequently it wakes up, the less time it spends in deep sleep
    a. Assuming time spent in active mode is estimated to be 2 seconds
    b. Asleep for the rest of the time

Average current draw:

$$I_{avg} = (duty\ cycle \times I_{active}) + ((1 - duty\ cycle) \times I_{sleep})$$

- $duty\ cycle = \frac{time\ spent\ in\ active\ mode}{total\ cycle\ time}$
- $I_{active} = 68mA = system\ draw\ during\ active\ mode$
- $I_{sleep} = 0.02016mA = system\ draw\ in\ deep\ sleep\ mode$

Table 3. Average Current Draw at varying wake up times

Frequency data collection, every:	**Duty cycle: Amount of time active in % form**	**Average Current Draw:** I_{avg} $I_{avg} = (duty\ cycle \times I_{active}) + ((1 - duty\ cycle) \times I_{sleep})$
10 minutes (600 sec)	$= \frac{2}{600} = 0.0033$ 0.33%	I_{avg} = (0.0033 x 68mA) + (0.9967 x 0.02016mA) = (0.2244) + (0.0201) = 0.244mA
5 minutes (300 sec)	$= \frac{2}{300} = 0.0067$ 0.67%	I_{avg} = (0.0067 x 68mA) + (0.9933 x 0.02016mA) = (0.456) + (0.0201) = 0.476mA
3 minutes (180 sec)	$= \frac{2}{180} = 0.0111$ 1.11%	I_{avg} = (0.0111 x 68mA) + (0.9889 x 0.02016mA) = (0.755) + (0.012) = 0.767mA
2 minutes (120 sec)	$= \frac{2}{120} = 0.0167$ 1.67%	I_{avg} = (0.0167 x 68mA) + (0.9833 x 0.02016mA) = (1.14) + (0.02) = 1.16mA
1 minute (60 sec)	$= \frac{2}{60} = 0.033$ 3.33%	I_{avg} = (0.033 x 68mA) + (0.967 x 0.02016mA) = (2.244) + (0.02) = 2.264 mA
30 seconds	$= \frac{2}{30} = 0.067$ 6.67%	I_{avg} = (0.067 x 68mA) + (0.933 x 0.02016mA) = (4.556) + (0.02) = 4.57mA
10 seconds	$= \frac{2}{10} = 0.2$ 20%	I_{avg} = (0.2 x 68mA) + (0.8 x 0.02016mA) = (13.6) + (0.016) = 13.62mA

$$battery\ life = \frac{battery\ capacity\ (mAh)}{Avg.\ Current\ Draw\ (mA)}$$

Table 4. Battery Life for each frequency

Frequency data collection, every:	**Battery Option 3: [2,500mAh]** - 3.7v 2,500mAh
10 minutes (600 sec)	$B_{life} = \frac{2{,}500mAh}{0.244mA} = 10,245.9\ hours$ || Days =10, 245 /24 = **426 days**
5 minutes (300 sec)	$B_{life} = \frac{2{,}500mAh}{0.476mA} = 5252\ hours$ || Days =5252/24 = **218 days**
3 minutes (180 sec)	$B_{life} = \frac{2{,}500mAh}{0.767mA} = 3259\ hours$ || Days =3259/24 = **135 days**
2 minutes (120 sec)	$B_{life} = \frac{2{,}500mAh}{1.16mA} = 2155\ hours$ || Days =2155/24 = **89 days**
1 minute (60 sec)	$B_{life} = \frac{2{,}500mAh}{2.264mA} = 1104\ hours$ || Days =1104/24 = **46 days**
30 seconds	$B_{life} = \frac{2{,}500mAh}{4.57mA} = 547\ hours$ || Days =547/24 = **22.7 days**
10 seconds	$B_{life} = \frac{2{,}500mAh}{13.62mA} = 183.5\ hours$ || Days =183. 5 /24 = **7.6 days**

Overall, the integration of power-saving modes remains a promising and impactful direction for future development. Successfully implementing deep sleep would not only improve system longevity but also demonstrate effective energy-aware embedded system design.

Replacing Ethernet with Wireless or Optical Interfaces

While Ethernet provided a reliable data transmission medium for our prototype, its requirement for physical cabling and access to network infrastructure limits deployment flexibility, particularly in wall or ceiling mounted installations. To transition our project to a market-ready product, a wireless or non-RF alternative will be considered to replace the W5500 Ethernet interface. For non-EMI-restricted environments, integrating Wi-Fi or Bluetooth directly into the receiver hub, leveraging the native wireless capabilities of the ESP32-S3 would eliminate the need for Ethernet entirely while preserving the fast, low-latency communication with the central console.

For EMI-sensitive settings where RF emissions must still be avoided, a non-RF alternative such as a USB tether or an optical-to-USB bridge could provide a wired, shielded data link to the central console without relying on Ethernet infrastructure. Another possible solution is the transition to a potential desktop dongle that could receive data optically through a photodiode interface, which would interface directly with the console via USB. These approaches would

simplify installation, expand compatibility with various physical environments and reduce user dependency on fixed port access issues in different institutes.

Replacing USB-C Receiver Power with a Battery

The receiver hub in its current form relies on USB-C for continuous power, which imposes practical limitations during installation. For example, the need to place the receiver near an outlet restricts mounting locations and adds visible wiring, which is undesirable in cleanroom or clinical environments. To improve flexibility and aesthetic integration, a battery-powered version of the receiver is proposed. This modification would mirror the one used in the transmitter hub, using a 3.7V lithium-ion battery regulated down to 3.3V for digital components and ±3V for the analog circuitry using the LM27762.

Implementing this change would involve redesigning the power subsystem on the receiver PCB to include safe battery charging, status monitoring, and mechanical housing for the battery cell - although this should be made relatively easier considering we have already gone through with these steps on the transmitter side. With battery integration, the receiver becomes fully untethered, capable of being installed in remote locations or mobile configurations without requiring any nearby power infrastructure. Additionally, by combining battery support with power optimization strategies like sleep-mode, the receiver has the potential to last for multiple weeks between charges, making it suitable for both temporary and long-term deployments.

Supporting Multiple Transmitters and/or Receivers

The current implementation of our project is built around a single transmitter-receiver pair. To scale the system for broader environmental coverage or multi-room monitoring, future iterations will need to support multiple transmitters and/or receivers operating together. This expansion would enable more complex installations in settings like large hospital wards, manufacturing cleanrooms, or smart buildings with multiple monitoring zones.

To support multiple transmitters, the system would implement data multiplexing strategies, either through time-division, frequency-division, or packet-based addressing. Along with this, the receiver would need updates to differentiate incoming data streams and assign them to the appropriate endpoints. The GUI and backend software would also need to support multiple streams of data along with visualization from multiple sources, ensuring that sensor data is accurately received and labeled.

One possibility is to perform one-time synchronization of multiple transmitter boards via WiFi and then perform time-division multiplexing to prevent multiple transmitter signals from interfering with each other at the receiver. Another possibility would be to place wide-angle LEDs on the receiver hub board (such as the TSAL6200) and receiver circuitry on the transmitter hub board to implement two-way communications and optical synchronization. However, further

design is needed to minimize the power consumption of the receiver circuitry on the transmitter hub boards.

Conclusion

Evaluation of Design Outcomes

Our project's system represents a strong, reliable and RFI-safe approach to real-time environmental monitoring, utilizing free-space optical communication to address challenges in RF sensitive settings like hospitals, laboratories, and cleanrooms. Through the integration of high-precision sensors, infrared data transmission, error correction algorithms, and a custom graphical interface, the prototype successfully met its core technical and performance goals. Real-world testing confirmed not only the viability of optical signaling at distances up to 3 meters (and with a limited budget and hence limited strength) but also the consistency of data transmission through noisy lighting conditions. The system operated with impressive latency and remained within its designed power envelope, even without the benefit of low-power optimization.

Beyond the metrics of performance, the system demonstrated strong cohesiveness between its subsystems. The receiver circuit proved capable of accurately restoring signals with minimal noise, and the UART-based firmware kept data integrity strong through our error correction methods. The use of standardized protocols, like UART, simplified the integration and has created opportunities for expansion. Likewise, the Python-based GUI served as a more than capable platform for both real-time visualization and long-term data logging.

Accuracy, Performance, and Lessons Learned

The final prototype achieved a high level of accuracy and reliability across its core sensing and communication functions, with most performance metrics meeting or exceeding initial expectations for a functional proof of concept. The sensor suite delivered consistent, real-time environmental readings, and the optical communication link preserved data structure and timing with little to no error. Through the use of Hamming codes, the system was able to automatically detect and correct single-bit transmission errors, ensuring robust performance even in the presence of analog signal distortion and ambient light interference.

Several lessons emerged over the course of development. The analog circuitry, particularly the high-gain transimpedance amplifier and comparator stage, demanded careful tuning and component selection to ensure signal reliability. Achieving stable power delivery to both analog and digital components required extensive decoupling and noise mitigation, especially when operating from a single USB source. Mechanically, precise optical alignment was essential, and future versions will benefit from adjustable fixtures. On the software side, the firmware architecture allowed for easy debugging and future extensibility, but more in depth optimization, especially with regard to power-saving modes, remains an important next step.

Overall, the project demonstrated that free-space optical data transmission is a viable and effective alternative to RF communication in environments where minimising RF interference is a priority. The foundation laid by this prototype supports future iterations that are more scalable, energy-efficient, and adaptive to complex deployment scenarios.

Appendices

Appendix A: Hardware PCB Schematics

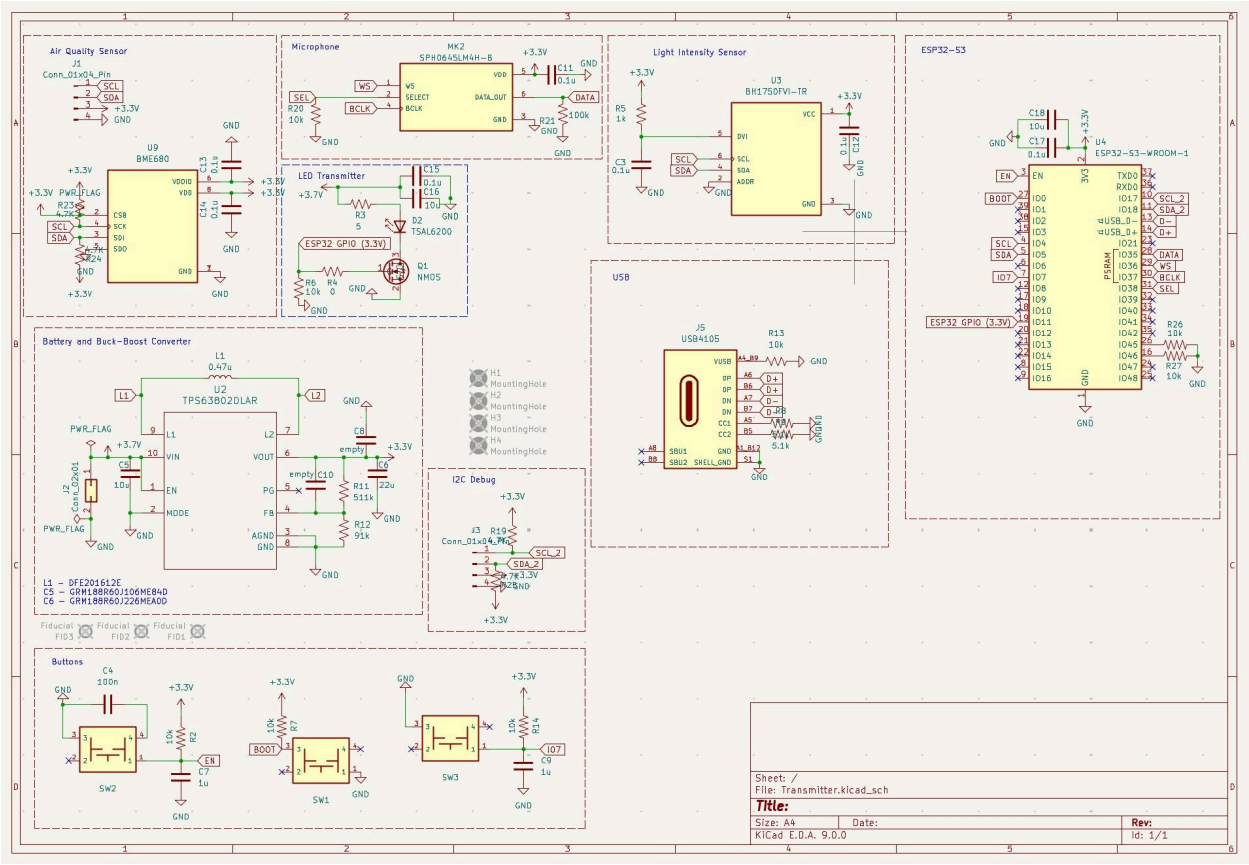

Figure A1. Transmitter PCB Schematic

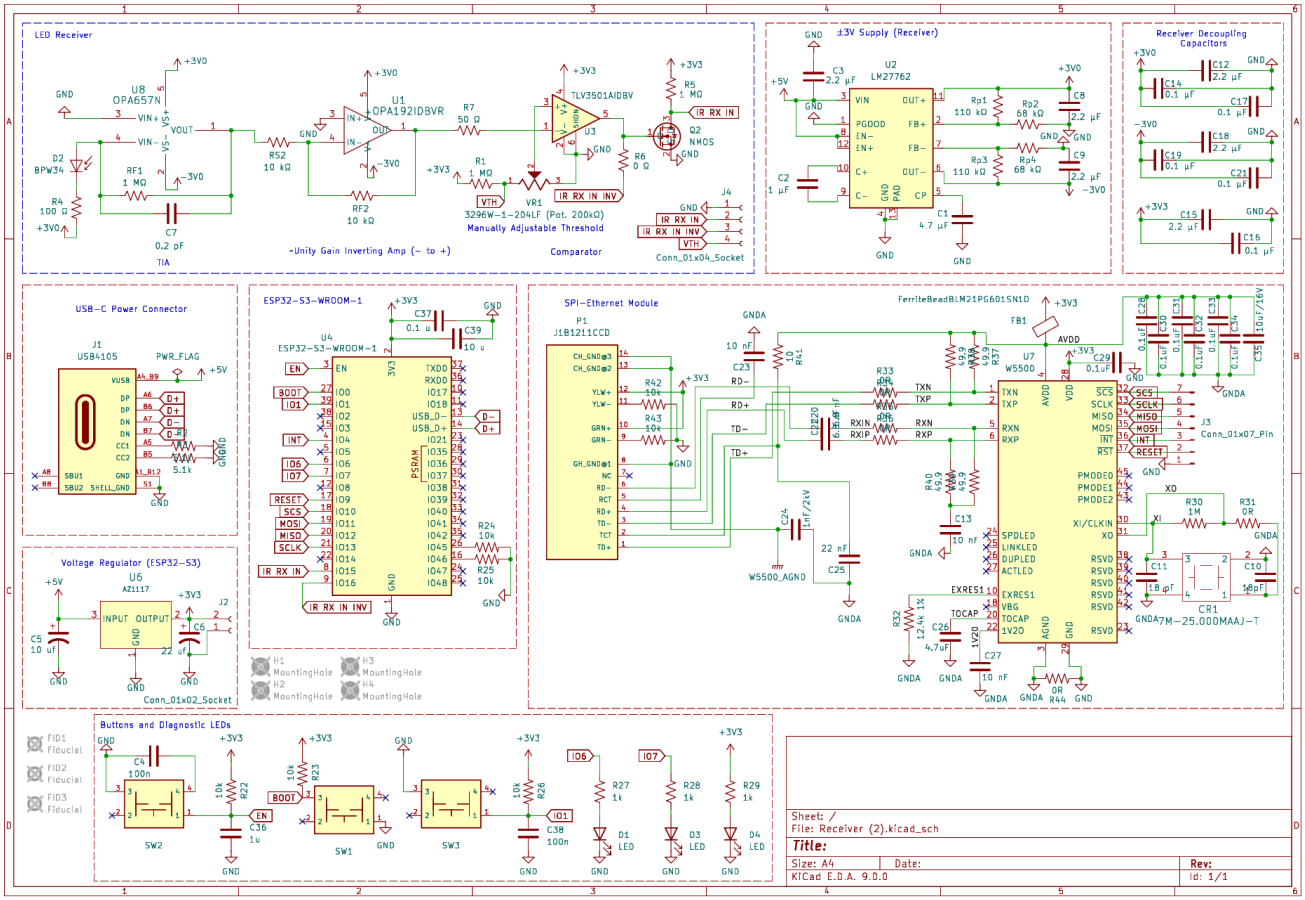

Figure A2. Receiver PCB Schematic

Appendix B: Complete Software Listings

Transmitter Code

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME680.h>
#include <BH1750.h>
#include <driver/i2s.h>
#include <Arduino.h>

// fix pin numbers
```

```
#define SDA_PIN 5
#define SCL_PIN 4
#define TX_PIN 11  // IR TX pin
// Define I²S GPIO pins based on our wiring
#define I2S_WS   36  // Word Select (LRCLK)
#define I2S_BCLK  37  // Bit Clock (BCLK)
#define I2S_DOUT  35  // Data Output (DOUT)

const int32_t MIC_MAX_AMPLITUDE = 8000000; // Approx max raw value from
mic
const int32_t MIC_MIN_THRESHOLD = 500;     // Noise floor (adjust based on
your room)

int x = 1;  // iteration tracker (only for test messages)

// voids
void BME_Setup();
void BH_Setup();
void BH_Measure();
void BME_Measure();
void MIC_Read();

// for hamming
// data to encode
char buffer[100];
// buffer to store encoded data for tx
char enc_buffer[200];

// function prototype for encoding function
void encode_hamming84(char*, int, char*);
void serial2_println_hamming84(String);

// I²S Configuration
void i2s_install() {
  const i2s_config_t i2s_config = {
      .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX), // Receive mode
      .sample_rate = 16000,  // 16 kHz sample rate
      .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,  // 32-bit data
      .channel_format = I2S_CHANNEL_FMT_ONLY_RIGHT,  // Read only the left
channel (L/R = GND)
```

```
      .communication_format = I2S_COMM_FORMAT_I2S,
      .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
      .dma_buf_count = 8,    // Number of buffers
      .dma_buf_len = 64,     // Buffer length
      .use_apll = false
  };

  i2s_driver_install(I2S_NUM_0, &i2s_config, 0, NULL);
}

void i2s_setpin() {
  const i2s_pin_config_t pin_config = {
      .bck_io_num = I2S_BCLK,
      .ws_io_num = I2S_WS,
      .data_out_num = I2S_PIN_NO_CHANGE,  // Not used for mic input
      .data_in_num = I2S_DOUT
  };

  i2s_set_pin(I2S_NUM_0, &pin_config);
}


// Create BME680 object using I2C
Adafruit_BME680 bme (&Wire);

// Create an instance of the BH1750 sensor
BH1750 lightMeter;

void setup() {
    Serial.begin(115200);
    while (!Serial);  // Wait for serial monitor to open
    // IR serial init (UART)
    Serial2.begin(115200, SERIAL_8N1, -1, TX_PIN, true);  // 'true'
enables inverted logic

    // Use a custom I2C bus (if needed)
    Wire.begin(SDA_PIN, SCL_PIN);
    BME_Setup();
    BH_Setup();
```

```
    pinMode(38, OUTPUT);
    digitalWrite(38,HIGH);

    i2s_install();
    i2s_setpin();
    i2s_start(I2S_NUM_0);

    // random seed init for artificially introducing bit error
    //randomSeed(analogRead(0));
}

void loop(){
  BME_Measure();
  BH_Measure();
  MIC_Read();

  // Tranmission code
  //Serial2.printf("Iteration: %d\n", x);
  Serial.println("Message Sent!");
  x += 1;
  delay(2000);  // Send every 0.2s
}

// functions

void BME_Setup()
{
  Serial.println("Initializing BME680 sensor...");

  if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME680 sensor, check
wiring!");
        while (1);
    }

   // Set up oversampling and filter settings
    bme.setTemperatureOversampling(BME680_OS_8X);
    bme.setHumidityOversampling(BME680_OS_2X);
    bme.setPressureOversampling(BME680_OS_4X);
    bme.setIIRFilterSize(BME680_FILTER_SIZE_3);
```

```
    bme.setGasHeater(320, 150); // 320°C for 150ms

}

void BH_Setup()
{
  Serial.println("Initializing BH1750 sensor...");

  if (lightMeter.begin(BH1750::CONTINUOUS_HIGH_RES_MODE)) {
    Serial.println("BH1750 sensor initialized successfully!");
  }
  else {
    Serial.println("Error: Could not find a valid BH1750 sensor, check
wiring!");
    while (1); // Halt the program if the sensor is not found
  }

  Serial.println("BH1750 sensor initialized!");
}

void BME_Measure()
{
  Serial.println("\nReading BME680 sensor data...");

    // Perform measurement
    if (!bme.performReading()) {
        Serial.println("Failed to perform reading :(");
        return;
    }

    // Print sensor values
    Serial.print("Temperature: ");
    Serial.print(bme.temperature);
    Serial.println(" °C");
    serial2_println_hamming84("Temperature: " + String(bme.temperature) +
" °C");

    Serial.print("Humidity: ");
    Serial.print(bme.humidity);
    Serial.println(" %");
```

```
    serial2_println_hamming84("Humidity: " + String(bme.humidity) + " %");

    Serial.print("Pressure: ");
    Serial.print(bme.pressure / 100.0);  // Convert to hPa
    Serial.println(" hPa");
    serial2_println_hamming84("Pressure: " + String(bme.pressure/100) + "
hPa");

    Serial.print("Gas Resistance: ");
    Serial.print(bme.gas_resistance / 1000.0);  // Convert to kΩ
    Serial.println(" kΩ");
    serial2_println_hamming84("Gas Resistance: " +
String(bme.gas_resistance / 1000.0) + " kΩ");
}
void BH_Measure()
{
  float lux = lightMeter.readLightLevel();  // Read light level in lux

    Serial.print("Light Intensity: ");
    Serial.print(lux);
    Serial.println(" lux");

    serial2_println_hamming84("Light Intensity: " + String(lux) + " lux");

    delay(2000);  // Wait 1 second before next reading
}

void MIC_Read()
{
  // Microphone Code
  int32_t sample_raw;
  size_t bytes_read;

  i2s_read(I2S_NUM_0, &sample_raw, sizeof(sample_raw), &bytes_read,
portMAX_DELAY);

  if (bytes_read > 0) {
      int32_t sample = sample_raw >> 8;

      // Sign extend
```

```
      if (sample & 0x00800000) {
          sample |= 0xFF000000;
      }

      // Calculate absolute value (audio "intensity")
      int32_t magnitude = abs(sample);

      if (magnitude < MIC_MIN_THRESHOLD) {
        magnitude = MIC_MIN_THRESHOLD;  // Use threshold as floor to avoid
log(0)
    }

    // Convert to dB (using MIC_MIN_THRESHOLD as reference)
    float sound_db = 20.0 * log10((float)magnitude / MIC_MIN_THRESHOLD);

    //Serial.printf("dB: %.2f\n", sound_db);

      // Display result
      Serial.print("Sound Level: ");
      Serial.printf("dB: %.2f dB\n", sound_db);
      serial2_println_hamming84("Sound Level: " + String(sound_db) +
"dB");
    }

}

void encode_hamming84(char* input_buffer, int input_buffer_len, char*
out_buffer) {
  unsigned char half, enc_half,p1,p2,p3,p4,d1,d2,d3,d4;
  int i2,j2;
  // cycle through input buffer bytes
  for (int i = 0; i < input_buffer_len; i++) {
    //i2 = i;
    // cycle through first and 2nd half of buffer bytes
    for (int j = 0; j < 2; j++) {
      //j2 = j;
      // 4 bit sequence to encode
      half = (buffer[i] >> j*4) & 0x0F;
      // data bits
      d1 = (half >> 3) & 1;
```

```
      d2 = (half >> 2) & 1;
      d3 = (half >> 1) & 1;
      d4 = half & 1;
      // calculate parity bit values
      p1 = d1 ^ d2 ^ d4;    // p1 = d1 ⊕ d2 ⊕ d4
      p2 = d1 ^ d3 ^ d4;    // p2 = d1 ⊕ d3 ⊕ d4
      p3 = d2 ^ d3 ^ d4;    // p3 = d2 ⊕ d3 ⊕ d4
      p4 = d1 ^ d2 ^ d3;    // p4 = d1 ⊕ d2 ⊕ d3
      out_buffer[i*2+j] = (p1 << 7) | (p2 << 6) | (d1 << 5) | (p3 << 4) |
(d2 << 3) | (d3 << 2) | (d4 << 1) | p4;
      // artificially introduce 1 bit error
      //out_buffer[i*2 + j] ^= (1 << random(0, 8));
    }
  }
  // sanity check for length
  //Serial.println(i2*2+j2+1);
}

void serial2_println_hamming84(String str1) {
  str1.toCharArray(buffer, sizeof(buffer));

  // Hamming 8,4 encoding
  // For segmentation of data: determine length of string in bytes
  int buffer_len = strlen(buffer);
  // encode and store encoded data in enc_buffer
  encode_hamming84(buffer, buffer_len, enc_buffer);
  // send encoded data over IR link
  Serial2.write(enc_buffer, buffer_len*2);
  Serial2.print("\n");
}
/*
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME680.h>
#include <BH1750.h>
#include <driver/i2s.h>
#include <Arduino.h>

// fix pin numbers
#define SDA_PIN 5
```

```
#define SCL_PIN 4
#define TX_PIN 11  // IR TX pin
// Define I²S GPIO pins based on our wiring
#define I2S_WS   36  // Word Select (LRCLK)
#define I2S_BCLK  37  // Bit Clock (BCLK)
#define I2S_DOUT  35  // Data Output (DOUT)

const int32_t MIC_MAX_AMPLITUDE = 8000000; // Approx max raw value from
mic
const int32_t MIC_MIN_THRESHOLD = 500;     // Noise floor (adjust based on
your room)

int x = 1;  // iteration tracker (only for test messages)

// voids
void BME_Setup();
void BH_Setup();
void BH_Measure();
void BME_Measure();

// for hamming
// data to encode
char buffer[100];
// buffer to store encoded data for tx
char enc_buffer[200];

// function prototype for encoding function
void encode_hamming84(char*, int, char*);
void serial2_println_hamming84(String);

// I²S Configuration
void i2s_install() {
  const i2s_config_t i2s_config = {
      .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX), // Receive mode
      .sample_rate = 16000,  // 16 kHz sample rate
      .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,  // 32-bit data
      .channel_format = I2S_CHANNEL_FMT_ONLY_RIGHT,  // Read only the left
channel (L/R = GND)
      .communication_format = I2S_COMM_FORMAT_I2S,
      .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
```

```
      .dma_buf_count = 8,    // Number of buffers
      .dma_buf_len = 64,     // Buffer length
      .use_apll = false
  };

  i2s_driver_install(I2S_NUM_0, &i2s_config, 0, NULL);
}

void i2s_setpin() {
  const i2s_pin_config_t pin_config = {
      .bck_io_num = I2S_BCLK,
      .ws_io_num = I2S_WS,
      .data_out_num = I2S_PIN_NO_CHANGE,  // Not used for mic input
      .data_in_num = I2S_DOUT
  };

  i2s_set_pin(I2S_NUM_0, &pin_config);
}


// Create BME680 object using I2C
Adafruit_BME680 bme (&Wire);

// Create an instance of the BH1750 sensor
BH1750 lightMeter;

void setup() {
    Serial.begin(115200);
    while (!Serial);  // Wait for serial monitor to open
    // IR serial init (UART)
    Serial2.begin(115200, SERIAL_8N1, -1, TX_PIN, true);  // 'true'
enables inverted logic

    // Use a custom I2C bus (if needed)
    Wire.begin(SDA_PIN, SCL_PIN);
    BME_Setup();
    BH_Setup();

    pinMode(38, OUTPUT);
    digitalWrite(38,HIGH);
```

```
    i2s_install();
    i2s_setpin();
    i2s_start(I2S_NUM_0);

    // random seed init for artificially introducing bit error
    //randomSeed(analogRead(0));
}

void loop(){
  BME_Measure();
  BH_Measure();

  // Microphone Code
  int32_t sample_raw;
  size_t bytes_read;

  i2s_read(I2S_NUM_0, &sample_raw, sizeof(sample_raw), &bytes_read,
portMAX_DELAY);

  if (bytes_read > 0) {
      int32_t sample = sample_raw >> 8;

      // Sign extend
      if (sample & 0x00800000) {
          sample |= 0xFF000000;
      }

      // Calculate absolute value (audio "intensity")
      int32_t magnitude = abs(sample);

      // Apply noise floor filtering
      if (magnitude < MIC_MIN_THRESHOLD) {
          magnitude = 0; // Treat anything below threshold as silence
      }

      // Map intensity to 1-1000 scale
      int scaled_value = map(magnitude, MIC_MIN_THRESHOLD,
MIC_MAX_AMPLITUDE, 1, 1000);
```

```
      // Clamp between 1 and 1000
      if (scaled_value < 1) scaled_value = 1;
      if (scaled_value > 1000) scaled_value = 1000;

      // Display result
      Serial.print("Sound Level: ");
      Serial.println(scaled_value);
      serial2_println_hamming84("Sound Level: " + String(scaled_value));
    }

  // Tranmission code
  //Serial2.printf("Iteration: %d\n", x);
  Serial.println("Message Sent!");
  x += 1;
  delay(200);  // Send every 0.2s
}

// functions

void BME_Setup()
{
  Serial.println("Initializing BME680 sensor...");

  if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME680 sensor, check
wiring!");
        while (1);
    }

   // Set up oversampling and filter settings
    bme.setTemperatureOversampling(BME680_OS_8X);
    bme.setHumidityOversampling(BME680_OS_2X);
    bme.setPressureOversampling(BME680_OS_4X);
    bme.setIIRFilterSize(BME680_FILTER_SIZE_3);
    bme.setGasHeater(320, 150); // 320°C for 150ms

}

void BH_Setup()
{
```

```
  Serial.println("Initializing BH1750 sensor...");

  if (lightMeter.begin(BH1750::CONTINUOUS_HIGH_RES_MODE)) {
    Serial.println("BH1750 sensor initialized successfully!");
  }
  else {
    Serial.println("Error: Could not find a valid BH1750 sensor, check
wiring!");
    while (1); // Halt the program if the sensor is not found
  }

  Serial.println("BH1750 sensor initialized!");
}

void BME_Measure()
{
  Serial.println("\nReading BME680 sensor data...");

    // Perform measurement
    if (!bme.performReading()) {
        Serial.println("Failed to perform reading :(");
        return;
    }

    // Print sensor values
    Serial.print("Temperature: ");
    Serial.print(bme.temperature);
    Serial.println(" °C");
    serial2_println_hamming84("Temperature: " + String(bme.temperature) +
" °C");

    Serial.print("Humidity: ");
    Serial.print(bme.humidity);
    Serial.println(" %");
    serial2_println_hamming84("Humidity: " + String(bme.humidity) + " %");

    Serial.print("Pressure: ");
    Serial.print(bme.pressure / 100.0);  // Convert to hPa
    Serial.println(" hPa");
```

```
    serial2_println_hamming84("Pressure: " + String(bme.pressure/100) + "
hPa");

    Serial.print("Gas Resistance: ");
    Serial.print(bme.gas_resistance / 1000.0);  // Convert to kΩ
    Serial.println(" kΩ");
    serial2_println_hamming84("Gas Resistance: " +
String(bme.gas_resistance / 1000.0) + " kΩ");
}
void BH_Measure()
{
  float lux = lightMeter.readLightLevel();  // Read light level in lux

    Serial.print("Light Intensity: ");
    Serial.print(lux);
    Serial.println(" lux");

    serial2_println_hamming84("Light Intensity: " + String(lux) + " lux");

    delay(2000);  // Wait 1 second before next reading
}

void encode_hamming84(char* input_buffer, int input_buffer_len, char*
out_buffer) {
  unsigned char half, enc_half,p1,p2,p3,p4,d1,d2,d3,d4;
  int i2,j2;
  // cycle through input buffer bytes
  for (int i = 0; i < input_buffer_len; i++) {
    //i2 = i;
    // cycle through first and 2nd half of buffer bytes
    for (int j = 0; j < 2; j++) {
      //j2 = j;
      // 4 bit sequence to encode
      half = (buffer[i] >> j*4) & 0x0F;
      // data bits
      d1 = (half >> 3) & 1;
      d2 = (half >> 2) & 1;
      d3 = (half >> 1) & 1;
      d4 = half & 1;
      // calculate parity bit values
```

```
      p1 = d1 ^ d2 ^ d4;   // p1 = d1 ⊕ d2 ⊕ d4
      p2 = d1 ^ d3 ^ d4;   // p2 = d1 ⊕ d3 ⊕ d4
      p3 = d2 ^ d3 ^ d4;   // p3 = d2 ⊕ d3 ⊕ d4
      p4 = d1 ^ d2 ^ d3;   // p4 = d1 ⊕ d2 ⊕ d3
      out_buffer[i*2+j] = (p1 << 7) | (p2 << 6) | (d1 << 5) | (p3 << 4) |
(d2 << 3) | (d3 << 2) | (d4 << 1) | p4;
      // artificially introduce 1 bit error
      //out_buffer[i*2 + j] ^= (1 << random(0, 8));
    }
  }
  // sanity check for length
  //Serial.println(i2*2+j2+1);
}

void serial2_println_hamming84(String str1) {
  str1.toCharArray(buffer, sizeof(buffer));

  // Hamming 8,4 encoding
  // For segmentation of data: determine length of string in bytes
  int buffer_len = strlen(buffer);
  // encode and store encoded data in enc_buffer
  encode_hamming84(buffer, buffer_len, enc_buffer);
  // send encoded data over IR link
  Serial2.write(enc_buffer, buffer_len*2);
  Serial2.print("\n");
}
*/
```

Receiver Code

```
#include <Arduino.h>
#include <SPI.h>
#include <Ethernet_Generic.h>
#include <EthernetUdp.h>  // UDP support

// Fixed MAC address for IT
byte mac[] = { 0x02, 0x08, 0xDC, 0x32, 0x19, 0xB7 };

// W5500 pin assignments
#define W5500_CS    10
```

```
#define W5500_RST   9
#define W5500_MISO  12
#define W5500_MOSI  11
#define W5500_SCK   13

#define RX_PIN 16  // Change this to your desired RX pin

// Target (desktop) IP and port — update as needed
IPAddress desktopIP(10, 37, 26, 178); // Replace with your desktop's IP
const unsigned int desktopPort = 4210;

EthernetUDP Udp;  // UDP object

void printMacAddress(byte *mac) {
  for (int i = 0; i < 6; i++) {
    if (mac[i] < 0x10) Serial.print("0");
    Serial.print(mac[i], HEX);
    if (i < 5) Serial.print(":");
  }
  Serial.println();
}

void setup() {
  Serial.begin(115200);
  Serial2.begin(115200, SERIAL_8N1, RX_PIN, -1, false);  // 'true' enables
inverted logic

  // Reset W5500
  pinMode(W5500_RST, OUTPUT);
  digitalWrite(W5500_RST, LOW);
  delay(50);
  digitalWrite(W5500_RST, HIGH);
  delay(200);

  // Start SPI
  SPI.begin(W5500_SCK, W5500_MISO, W5500_MOSI, W5500_CS);
  Ethernet.init(W5500_CS);

  Serial.println("Starting Ethernet with DHCP...");
  Ethernet.begin(mac);  // Use fixed MAC address
```

```
  Serial.print("MAC Address: ");
  printMacAddress(mac);

  Serial.print("Assigned IP: ");
  Serial.println(Ethernet.localIP());

  if (Ethernet.linkStatus() == LinkON) {
    Serial.println("✅ Ethernet cable detected.");
  } else {
    Serial.println("⚠️ Ethernet cable not detected.");
  }

  // Start UDP
  Udp.begin(4210);  // Port to listen on if needed
}

void loop() {
  const char* msg = "Hello from ESP32 via UDP!";
  Udp.beginPacket(desktopIP, desktopPort);
  Udp.write(msg);
  Udp.endPacket();

  Serial.print("Sent message to ");
  Serial.print(desktopIP);
  Serial.print(":");
  Serial.println(desktopPort);

  delay(1000);  // Send every second

  if (Serial2.available()) {
    String receivedData = Serial2.readStringUntil('\n');
    Serial.print("Received: ");
    Serial.println(receivedData);
}
}
```

RX Aim Code

```
#include <Arduino.h>

#define SENSE_PIN 15
#define LED_PIN 7

void setup() {
  pinMode(SENSE_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(digitalRead(SENSE_PIN) == HIGH) {
    digitalWrite(LED_PIN, HIGH);
  }
  else {
    digitalWrite(LED_PIN, LOW);
  }
}
```

TX Aim Code

```
#include <Arduino.h>

#define LED 11 // led on pin 15

void setup() {
  pinMode(LED,OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
```

```
  digitalWrite(LED, HIGH);      // Turn LED on
  delay(1000);                  // Delay 0.25s
  digitalWrite(LED, LOW);      // Turn LED off
  delay(1000);                  // Delay 0.25s
}
```

NEXASENSEE GUI Code

```
import sys
import socket
import pandas as pd
from PyQt5.QtWidgets import (
    QApplication, QWidget, QLabel, QPushButton, QVBoxLayout, QHBoxLayout,
    QGroupBox, QCheckBox, QDialog, QTimeEdit, QFileDialog, QMessageBox
)
from PyQt5.QtCore import QTimer, QDateTime, QTime
from PyQt5.QtGui import QFont

class SensorGUI(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("NEXASENSEE Central Console")
        self.showMaximized()
        self.setStyleSheet("background-color: #1E1E1E; color: #E0E0E0;")

        self.sensor_labels = {}
        self.data_log = []

        main_layout = QHBoxLayout()
        left_layout = QVBoxLayout()

        # Sensor display boxes with units in titles
        self.sensors = {
            "Temperature (°C)": "TEMP",
            "Humidity (%)": "HUMIDITY",
            "Pressure (hPa)": "PRESSURE",
            "Light Intensity (lux)": "LIGHT",
            "Gas (Ω)": "GAS",
            "Sound (dB)": "SOUND"
        }
```

```
        for sensor_display_name in self.sensors:
            box = QGroupBox(sensor_display_name)
            box.setStyleSheet("QGroupBox { border: 2px solid #0078D7;
border-radius: 10px; padding: 10px; font-weight: bold; }")
            layout = QVBoxLayout()
            label = QLabel("Waiting for data...")
            label.setFont(QFont("Arial", 12))
            layout.addWidget(label)
            box.setLayout(layout)
            self.sensor_labels[sensor_display_name] = label
            left_layout.addWidget(box)

        # Record Data button
        self.record_button = QPushButton("Record Data")
        self.record_button.setStyleSheet("QPushButton { background-color:
#0078D7; color: white; font-size: 14px; padding: 10px; border-radius: 5px;
}")
        self.record_button.clicked.connect(self.open_recording_window)
        left_layout.addWidget(self.record_button)

        main_layout.addLayout(left_layout)
        self.setLayout(main_layout)

        # Set up UDP socket
        self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.udp_socket.bind(("0.0.0.0", 4210))
        self.udp_socket.setblocking(False)

        # Set up timer to check for UDP data
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_sensor_data)
        self.timer.start(500)

        def update_sensor_data(self):
        try:
            data, addr = self.udp_socket.recvfrom(1024)
            line = data.decode(errors="ignore").strip()  # Ignore bad
UTF-8
            print(f"Received from {addr}: {line}")
            pairs = line.split(",")
```

```
            for pair in pairs:
                if ":" not in pair:
                    continue  # Skip malformed entries

                try:
                    key, value = pair.split(":", 1)
                    key = key.strip().upper()
                    value = value.strip()

                    for sensor_name, keyword in self.sensors.items():
                        if keyword in key:
                            self.sensor_labels[sensor_name].setText(value)
                except Exception as inner_err:
                    print(f"⚠️ Error processing pair '{pair}':
{inner_err}")

        except BlockingIOError:
            pass  # No new data yet
        except Exception as e:
            print(f"❌ Error in update_sensor_data: {e}")


    def open_recording_window(self):
        self.record_window = RecordingWindow(self)
        self.record_window.setStyleSheet("background-color: #2E2E2E;
color: #E0E0E0;")
        self.record_window.show()

class RecordingWindow(QDialog):
    def __init__(self, main_window):
        super().__init__()
        self.setWindowTitle("Data Recording Settings")
        self.setGeometry(200, 200, 400, 300)
        self.main_window = main_window
        self.setStyleSheet("background-color: #2E2E2E; color: #E0E0E0;")

        layout = QVBoxLayout()
        self.checkboxes = {}
        for sensor in main_window.sensor_labels.keys():
```

```
            checkbox = QCheckBox(sensor)
            layout.addWidget(checkbox)
            self.checkboxes[sensor] = checkbox

        self.duration_box = QTimeEdit()
        self.duration_box.setDisplayFormat("HH:mm:ss")
        self.duration_box.setTime(QTime(0, 0, 10))
        layout.addWidget(self.duration_box)

        self.select_path_button = QPushButton("Select Save Location")
        self.select_path_button.setStyleSheet("QPushButton {
background-color: #444; color: white; padding: 6px; }")
        self.select_path_button.clicked.connect(self.select_file_path)
        layout.addWidget(self.select_path_button)

        self.start_button = QPushButton("Start Recording")
        self.start_button.setStyleSheet("QPushButton { background-color:
#0078D7; color: white; font-size: 14px; padding: 10px; border-radius: 5px;
}")
        self.start_button.clicked.connect(self.start_recording)
        layout.addWidget(self.start_button)

        self.setLayout(layout)
        self.file_path = ""
        self.timer = None
        self.recorded_data = []

    def select_file_path(self):
        path, _ = QFileDialog.getSaveFileName(self, "Save File", "",
"Excel Files (*.xlsx);;CSV Files (*.csv)")
        if path:
            self.file_path = path
            self.select_path_button.setText(f"Save to: {path}")

    def start_recording(self):
        selected_sensors = [sensor for sensor, checkbox in
self.checkboxes.items() if checkbox.isChecked()]
        if not selected_sensors:
            QMessageBox.warning(self, "No Sensors Selected", "Please
select at least one sensor to record.")
```

```
            return

        duration = self.duration_box.time()
        total_seconds = duration.hour() * 3600 + duration.minute() * 60 +
duration.second()

        if not self.file_path or (not self.file_path.endswith(".csv") and
not self.file_path.endswith(".xlsx")):
            QMessageBox.warning(self, "Invalid File Name", "Select a valid
file path ending with .csv or .xlsx.")
            return

        self.recorded_data = []
        self.timer = QTimer(self)
        self.timer.timeout.connect(lambda:
self.record_sensor_data(selected_sensors))
        self.timer.start(1000)
        QTimer.singleShot(total_seconds * 1000, self.stop_recording)

        QMessageBox.information(self, "Recording Started", f"Recording
{selected_sensors} for {total_seconds} seconds.")

    def record_sensor_data(self, selected_sensors):
        timestamp = QDateTime.currentDateTime().toString("yyyy-MM-dd
HH:mm:ss")
        for sensor in selected_sensors:
            value =
self.main_window.sensor_labels[sensor].text().split()[0]
            self.recorded_data.append({"timestamp": timestamp, "sensor":
sensor, "value": value})

    def stop_recording(self):
        if self.timer:
            self.timer.stop()

        try:
            df = pd.DataFrame(self.recorded_data)
            if self.file_path.endswith(".csv"):
                df.to_csv(self.file_path, index=False)
            else:
```

```
                df.to_excel(self.file_path, index=False)
            QMessageBox.information(self, "Recording Complete", f"Data
saved to '{self.file_path}'.")
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to save
file:\n{str(e)}")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = SensorGUI()
    window.show()
    sys.exit(app.exec_())
```

Appendix C: Relevant Component Datasheets

- ESP32-S3 (Microcontroller)
- BME680 (Air sensor)
- SPH0645LM4H-B (Microphone)
- BH1750 (Light intensity sensor)
- WIZnet 5500 (Ethernet chip)
    - Design Guide 1
    - Design Guide 2
- TPS63802DLAR (Buck-Boost Converter)
- LM27762DSST (Charge pump IC for ±3 V dual supply)
- 3.7V 2500mAh Lithium-ion battery
- BPW34 (IR Photodiode)
- VSLY5940 (940 nm IR LED)
- OPA657N/250 (Op-Amp used in receiver TIA stage)
- OPA192IDBVR (Op-Amp used in receiver unity gain inverting amplifier stage)
- TLV3501AIDBVR (Comparator used in receiver circuitry)

Appendix D: Background and Theory References

[1] *What causes electromagnetic interference?*. Compliance Testing. (2025, February 4). https://compliancetesting.com/what-causes-electromagnetic-interference/

[2] *EMC Certification Guide: FCC, CE & Other Compliance Marks*. Compliance Testing. (2025a, April 3). https://compliancetesting.com/emc-certification/#:~:text=FCC%20Part%2015%20ensures%20that,DoC)

[3] Gökmen, N., Erdem, S., Toker, K. A., Öçmen, E., Gökmen, B. I., & Özkurt, A. (2016, October 1). *Analyzing exposures to electromagnetic fields in an Intensive Care Unit*. Turkish journal of anaesthesiology and reanimation. https://pmc.ncbi.nlm.nih.gov/articles/PMC5118007/