AUDIO TRANSMISSION USING LIGHT FIDELITY (LiFi) TECHNOLOGY

Report for ESP300

By

Group 2A

Harsh Mittal :

Preksha Mishra : 2022ES11849

Jairam

Navya Tripathi

Jaivardhan

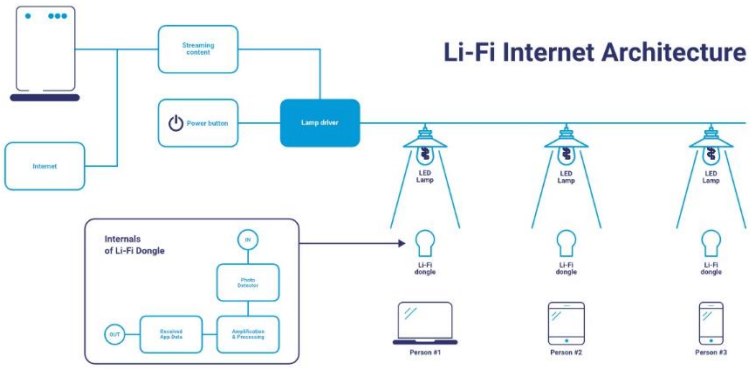
Dated: 24th April, 2025

Table of Contents

1. Abstract

The objective of this project is to design and implement a functional prototype for audio transmission using Li-Fi (Light Fidelity) technology. By leveraging visible light communication (VLC), this system demonstrates the wireless transmission of pre-recorded audio signals through modulated LED light, eliminating the need for traditional RF-based systems. The goal is to achieve a simple yet efficient unidirectional communication link capable of transmitting audio files from a computer to a receiver using basic hardware components and a custom Python-based software interface. The project emphasizes accurate audio transfer, minimal signal distortion, and cost-effective implementation, serving as a foundation for future innovations in Li-Fi-based communication systems.

Fig. 1: Typical Li-Fi Data Transmission Architecture [1]



2. List of Components

Category	Component	Description
Transmitting Side	LED (High Brightness, 3.3V Compatible)	For light-based data transmission.
	USB to UART Converter (CP2102)	For serial communication between PC and LED.
	Arduino Uno Controller	For controlling UART output.
	Resistors (e.g., 220Ω)	For current limiting to protect LED.
	Breadboard and Jumper Wires	For prototyping and connections.
	9V Battery + Clip Connector	Power source for the transmitting side.
Receiving Side	Photodiode (SFH-213-FA)	For detecting modulated light signals.
	Transimpedance Amplifier Circuit	Converts photodiode current to voltage.
	LM386 Audio Amplifier Board	To amplify the audio signal.
	Capacitors (e.g., 1000µF, 100µF)	For power filtering and amplifier coupling.
	10k Potentiometer	For volume control.
	Speaker	For audio output playback.
	Breadboard and Jumper Wires	For connections on the receiver side.

3. Architecture

Transmitting Side:

Source Computer: The computer acts as the central processing unit, where the audio file (e.g., WAV or MP3) is processed and converted into a serial data stream for transmission. Python scripts are used to break the audio into manageable packets that can be sent via UART communication.

USB to UART Converter (CP2102): This component enables serial communication between the computer and the microcontroller. It converts the digital data from the computer into a serial format that the Arduino can understand and transmit via UART.

Arduino Uno: The Arduino microcontroller is responsible for controlling the transmission process. It receives the serial data from the USB-to-UART converter and modulates it into light pulses, which are used to represent the binary data. The Arduino controls the LED, flashing it on and off in accordance with the data.

LED (High Brightness, 3.3V Compatible): This LED is used for light-based transmission of the audio data. The Arduino controls the LED to emit light pulses corresponding to the audio data, effectively sending the information via LiFi.

Resistors: These are used to limit the current flowing through the LED, ensuring that it operates within safe limits and doesn't get damaged.

Power Supply (9V Battery): The power supply ensures that both the Arduino and the LED receive the necessary voltage for operation. A 9V battery powers these components, and the clip connector makes it easy to connect and disconnect the power supply.

Receiving Side:

Receiving Computer: The receiving computer acts as the decoder for the transmitted audio data. It captures the light signal using a photodiode, decodes the data, and plays the audio through its internal speakers or audio output system.

Photodiode (SFH-213-FA): The photodiode is responsible for detecting the modulated light signal emitted by the LED. It converts the incoming light pulses into an electrical signal, which is then processed by the receiving computer.

Receiving Software: Similar to the transmitting side, software running on the receiving computer decodes the electrical signal from the photodiode. The decoded data is then converted back into the original audio format (e.g., WAV or MP3), and the audio is played.

Key Aspects of the Architecture:

No Complex Modulation: The system uses a simple protocol where the audio data is directly converted into light pulses. Both the transmitting and receiving sides use the same protocol, meaning there is no need for complex modulation techniques. The data is transmitted in packets of binary information that the receiving side can directly decode.

One-Way Audio Transmission: The system is designed for one-way audio transmission. The audio file is processed on the source device, transmitted via light pulses, and then decoded and played on the destination device.

LiFi Technology: The system utilizes Light Fidelity (LiFi), where the LED acts as the light source that transmits the data. The photodiode at the receiving side captures the modulated light and converts it back to an electrical signal. This enables audio data to be transmitted wirelessly via visible light.

4. Software

Transmitting Python code:

```
import wave
import numpy as np
import serial
import time

# CONFIG
FILE_PATH = "happy_birthday.wav"
SERIAL_PORT = "COM4"
BAUD_RATE = 115200
REPEAT_COUNT =20  # Number of times to repeat the transmission
# Load WAV audio
with wave.open(FILE_PATH, 'rb') as wf:
    sampwidth = wf.getsampwidth()
    n_frames = wf.getnframes()
    audio_data = wf.readframes(n_frames)

# Convert to 8-bit values
audio_array = np.frombuffer(audio_data, dtype=np.uint8 if sampwidth == 1 else np.int16)
if sampwidth > 1:
    audio_array = (((audio_array.astype(np.int32)) + 32768) // 256).astype(np.uint8)

# Open serial connection
ser = serial.Serial(SERIAL_PORT, BAUD_RATE)
time.sleep(2)  # Give Arduino time to reset

# Send audio data multiple times
for repeat in range(REPEAT_COUNT):
    print(f"Transmission round {repeat + 1}")
    for byte in audio_array:
        ser.write(bytes([byte]))
        # time.sleep(0.001)  # Uncomment if needed for slow devices

ser.close()
print("Transmission complete.")
```

Code Explanation:

This Python script facilitates the transmission of audio data over a serial connection for Li-Fi-based communication. It begins by loading a `.wav` audio file and reading its raw frame data using (`wave.open`). The sample width and total number of frames are extracted, and the audio is converted into 8-bit values using (`np.frombuffer`) and type conversion logic, to ensure compatibility with most microcontrollers.

A serial connection is established using (`serial.Serial`), followed by a short delay (`time.sleep`) to allow the microcontroller (e.g., Arduino) to reset. The script then repeatedly sends the byte-formatted

audio data over the serial interface inside a loop, controlled by the configured `REPEAT_COUNT`. Each byte is sent using (`ser.write`), creating a simple yet effective data transmission system.

Finally, the serial port is closed with (`ser.close`) after all rounds of transmission are completed. The overall process supports the transmission of digital audio via UART, which is essential for modulating and transmitting the signal through an LED in the Li-Fi system.

Transmitting Arduino Uno Code:

```
#define PWM_PIN_1 9            // OOK output 1 (e.g., to LED)
#define PWM_PIN_2 18           // OOK output 2 (TX1 pin as digital)
#define SAMPLE_INTERVAL 50     // 20 kHz bit rate (50 µs per bit)

void setup() {
  Serial.begin(921600);        // USB Serial for receiving from Python
  pinMode(PWM_PIN_1, OUTPUT);
  pinMode(PWM_PIN_2, OUTPUT);
}

void loop() {
  if (Serial.available()) {
  byte val = Serial.read();   // Receive one byte
  transmitByteOOK(val);       // Send same OOK on both pins
  }
}

void transmitByteOOK(byte val) {
  for (int b = 0; b < 8; b++) {
  bool bit = (val >> (7 - b)) & 1;
  digitalWrite(PWM_PIN_1, bit ? HIGH : LOW);
  digitalWrite(PWM_PIN_2, bit ? HIGH : LOW);
  delayMicroseconds(SAMPLE_INTERVAL);
  }
}
```

Code Explanation:

This Arduino code acts as the core hardware-level component in the visible light communication (Li-Fi) system by handling the modulation of incoming audio data into light pulses using On-Off Keying (OOK). It works in tandem with the Python-based transmission script running on a PC or similar host device, which reads an audio file (WAV format), processes it into 8-bit chunks, and transmits these byte-by-byte over a serial USB connection to the Arduino.

The Arduino initializes serial communication at a high baud rate of 921600 bps in the setup() function to match the sender's transmission speed. It also sets two digital output pins (PWM_PIN_1 and PWM_PIN_2) as outputs, which are connected to high-brightness LEDs or an LED driver circuit. These pins will serve as the OOK modulators — meaning, the LEDs will be turned on or off based on whether each bit in the audio byte is a '1' or a '0'.

In the main loop, the Arduino continuously checks for incoming data (Serial.available()). When it detects that data is available, it reads a byte (Serial.read()) and immediately passes it to the

transmitByteOOK() function. This function dissects the byte into its individual bits using bitwise operations, starting from the most significant bit (MSB). For each bit, it sets both output pins HIGH if the bit is 1 and LOW if the bit is 0, holding each state for a fixed duration of 50 microseconds (SAMPLE_INTERVAL). This creates a bitstream of light pulses representing the original audio signal. The consistent timing ensures a stable bit rate of 20 kHz, which is sufficient for transmitting low-fidelity audio over short-range optical links.

Receiving ESP32 Code:

```
#define SIGNAL_PIN 34  // GPIO for input from amplifier

#define SAMPLE_INTERVAL_MICROS 50  // 20 kHz = 50 µs

unsigned long last_sample_time = 0;


void setup() {

 Serial.begin(921600);  // High baud rate

 pinMode(SIGNAL_PIN, INPUT);

}
```

```
void loop() {

 unsigned long now = micros();

 if (now - last_sample_time >= SAMPLE_INTERVAL_MICROS) {

  last_sample_time = now;

  int bit = digitalRead(SIGNAL_PIN);

  Serial.write(bit ? 1 : 0);  // Send bit as byte: 0x01 or 0x00

 }
```

Code Explanation:

The ESP32 decodes the signal into digital bits and sends them via a serial connection to a computer or another device. The code is set up to handle high-speed sampling and bit extraction.

The setup() function initializes the serial communication with a high baud rate (921600) to match the rate used by the Arduino and ensure fast data transmission. It also configures the signal input pin (SIGNAL_PIN) as an input pin to receive the modulated signal from the amplifier, which amplifies the modulated light signal detected by the photodiode.

In the loop() function, the system continuously checks the elapsed time since the last sample using micros() (which returns the number of microseconds since the program started). This is done to ensure that sampling happens at the defined interval (SAMPLE_INTERVAL_MICROS), corresponding to a 20 kHz bit rate (50 µs per bit).

Once the appropriate time has passed, the code samples the signal pin (`SIGNAL_PIN`) by reading its digital state with digitalRead(). The state of the signal pin corresponds to the received bit: a HIGH state

represents a logical "1," and a LOW state represents a logical "0." The Serial.write() function then sends this bit to the serial output, formatted as a byte (either 0x01 for "1" or 0x00 for "0").

This process continues in real time, where the received data is decoded bit by bit and transmitted over serial to the connected device (e.g., a computer). The result is the successful reception of the light-modulated data transmitted by the Arduino, which can then be processed or played as audio on the receiving side. The use of micros() ensures that the sampling is done with precise timing, maintaining the integrity of the transmitted signal.

Receiving Python Code:

```
import serial
import time
import numpy as np
import wave
import sounddevice as sd


# --- CONFIG ---
COM_PORT = 'COM3'        # Replace with your actual COM port
BAUD_RATE = 115200       # Match your transmitter's baud rate
SAMPLE_RATE = 44100      # Match your original WAV sample rate
DURATION = 5             # Duration to receive (in seconds)


# --- SERIAL SETUP ---
ser = serial.Serial(COM_PORT, BAUD_RATE)
print(f"Connected to {COM_PORT}")
time.sleep(2)  # Allow device to initialize


bit_buffer = ''
samples = []


print("Receiving OOK bits...")


start_time = time.time()
while time.time() - start_time < DURATION:
    if ser.in_waiting:
        raw_byte = ser.read(1)
        bit_val = ord(raw_byte)
if bit_val in [0, 1]:
            bit_buffer += str(bit_val)
            if len(bit_buffer) == 8:
                byte_val = int(bit_buffer, 2)
                samples.append(byte_val)
                bit_buffer = ''


ser.close()
print(f"Received {len(samples)} samples.")


# --- NORMALIZE AND PLAY AUDIO ---
audio = np.array(samples, dtype=np.uint8)
audio_normalized = (audio.astype(np.float32) - 128) / 128  # Normalize to [-1, 1]


sd.play(audio_normalized, samplerate=SAMPLE_RATE)
sd.wait()
```

```
# --- SAVE TO WAV ---
with wave.open("reconstructed_audio.wav", 'wb') as wf:
    wf.setnchannels(1)
    wf.setsampwidth(1)  # 8-bit audio
    wf.setframerate(SAMPLE_RATE)
    wf.writeframes(audio.tobytes())

print("Audio saved as reconstructed_audio.wav)
```

Code Explanation:

The Python code establishes a serial communication link to receive OOK (On-Off Keying) modulated data transmitted from a device. The data is received bit by bit, with each byte decoded and stored in a list of samples. Once the data is collected, it is normalized to match audio playback standards, converting the raw 8-bit data into a float format suitable for sound output.

The program uses `sounddevice` to play the normalized audio data in real time, allowing immediate feedback of the received transmission. It also saves the received data as a WAV file, ensuring that the audio can be stored for later analysis or playback. The process continues for a specified duration, after which the program terminates, having successfully handled the reception, playback, and storage of the transmitted audio.

This demonstrates a simple yet effective way of receiving, decoding, and processing modulated audio signals over serial communication for a Li-Fi-based system.

5. Applications

The developed Li-Fi audio transmission prototype demonstrates the practical feasibility of using visible light for real-time wireless audio communication. Its applications span multiple domains where conventional radio frequency (RF)-based communication is either restricted, unreliable, or undesirable. Key application areas include:

Indoor Audio Distribution Systems: This prototype can be applied in venues such as museums, galleries, or educational spaces, where audio content is broadcasted through LEDs. The system transmits audio without the need for conventional wireless devices like headphones or speakers. By utilizing LED light for both illumination and communication, it reduces energy consumption compared to traditional wireless methods like Bluetooth and Wi-Fi. Moreover, its directional transmission ensures efficient, interference-free audio delivery, optimizing power usage.

Energy-Efficient Smart Lighting Networks: Li-Fi's integration with smart lighting systems offers a dual-purpose solution, combining illumination with high-speed data transmission. This application is particularly useful in energy-conscious environments like smart homes or offices, where both lighting and communication needs are met by the same infrastructure. By leveraging the same LED lights for data transfer, the system significantly reduces the need for additional communication devices, contributing to overall energy savings and supporting the growth of energy-efficient IoT networks.

Point-to-Point Communication in RF-Restricted Environments: Li-Fi offers inherent security benefits due to its reliance on visible light, which cannot penetrate opaque barriers like walls. This makes it highly suitable for secure communication in environments such as hospitals or military zones, where RF communication is restricted. Additionally, it can be deployed in disaster zones where traditional RF networks may fail, providing a temporary communication solution using off-grid LED lighting setups.

Unique Selling Point (USP)

The developed Li-Fi prototype showcases a minimal, low-cost solution for audio and byte-level optical communication using visible light. By leveraging On-Off Keying (OOK) and basic serial data handling, the system sidesteps the need for expensive optics, precision synchronization, or advanced modulation schemes. While OOK introduces susceptibility to noise and limits real-time audio fidelity, these trade-offs are precisely what make the prototype valuable in constrained and cost-sensitive environments — where reliability, EMI immunity, and deployment simplicity are more important than speed or resolution.

Purpose-Built for Low-Bandwidth, High-Noise Scenarios: In RF-restricted zones such as hospitals, defense installations, or industrial floors with heavy EMI, traditional wireless communication becomes either infeasible or risky. Here, the prototype provides an optically-isolated, low-bandwidth channel ideal for periodic data transfer or burst-mode communication. The system's noise tolerance is acceptable in these use-cases where minor packet drops do not compromise the core application.

Offline and Infrastructure-Free Deployment: Unlike commercial optical communication systems that depend on synchronized clocks, complex PHY layers, or cloud integration, this system operates entirely offline using commodity microcontrollers and basic analog electronics. Its simplicity makes it deployable in remote, disaster-struck, or infrastructurally fragile areas — particularly for one-way transmission of alerts, pre-recorded messages, or configuration data using portable LED-based sources.

Better Than Overengineered Alternatives — For the Right Job: Commercial Li-Fi systems and optical modems prioritize bandwidth, security layers, and mesh networking — making them expensive,

overbuilt, and poorly suited for educational, humanitarian, or grassroots tech deployments. In contrast, this prototype demonstrates how controlled light sources can transmit digital data or compressed audio between devices without RF, Bluetooth, Wi-Fi, or costly optical hardware. Its clear line-of-sight constraint becomes an advantage in controlled physical environments, offering spatial security and eliminating cross-talk.

Ideal for Education and Field Prototypes: The modular architecture — combining basic microcontrollers, amplifiers, and photodiodes — makes the system replicable for classroom demonstrations, student projects, or public tech exhibitions. Its visual transparency (you see the LED flicker and know data is moving) offers intuitive feedback, making it uniquely suited for communicating the core principles of optoelectronics, bitstreaming, and physical-layer data transfer to beginners.

Ultimately, the prototype doesn't aim to compete with high-end Li-Fi platforms — it complements them in the niches they ignore. Where others chase speed and scale, this system prioritizes resilience, cost-

accessibility, and deployment in edge environments where mainstream tech fails or overcomplicates the problem.

6. Learning Outcomes

This group project provided a multidimensional learning experience, combining theoretical knowledge with hands-on system design and implementation. Through iterative experimentation and collaborative problem-solving, we achieved the following key outcomes:

End-to-End System Integration Skills: We gained a practical understanding of how to integrate hardware (LEDs, photodiodes, amplifiers) with software components (Python-based data slicing, serial transmission) to create a working communication system. This end-to-end exposure enhanced our ability to think in terms of system architecture, timing constraints, and real-world signal integrity challenges.

Embedded Systems and Serial Communication: Working with Arduino and UART protocols allowed us to develop a firm grasp of embedded communication pipelines. We understood how baud rate, packet size, and buffer synchronization affect real-time data transfer, especially over unconventional channels like light.

Audio Processing and Frame Handling: We learned to handle and process WAV audio data programmatically. This included frame-wise slicing, packet framing with start-of-frame markers, and implementing padding strategies — all of which are essential concepts in digital communication.

Signal Conditioning and Photodetection: The receiver-side work taught us about the principles of photodiode behaviour, current-to-voltage conversion using transimpedance amplifiers, and signal amplification using audio ICs. This reinforced our understanding of analog electronics and noise filtering in low-signal environments.

Multithreading and Queued Processing in Python: By implementing multiprocessing and inter-process communication via queues, we learned to manage concurrent tasks — one for slicing and one for transmitting — which is essential for any real-time application development.

Team Collaboration and Agile Prototyping: We practiced agile development, iteratively testing and modifying both hardware and software components. Regular group discussions helped us troubleshoot issues, divide responsibilities effectively, and maintain clear documentation — key aspects of any collaborative engineering effort.

Critical Thinking and Innovation: Finally, we learned how to adapt standard tools and protocols in creative ways — such as repurposing UART over visible light — to solve real-world problems. This process honed our ability to innovate within practical constraints and to evaluate trade-offs between complexity, cost, and performance.