Project 1

Odin Augustine Erik Lundstam,
Rajvir Singh, James Guentert

BME 60A | Dr. Daryl Preece | 10am | 2/7/2025

Background Research: LiFi transmission

Electromagnetic (EM) communication is the most widely adopted form of wireless transmission, utilizing frequencies of 2.4 GHz, 5 GHz, and 6 GHz in technologies such as Wi-Fi, Bluetooth, and cellular networks. Hospitals and medical facilities rely on Wi-Fi to transmit sensitive information and coordinate operations, while growing consumer demand for high-speed data places increasing strain on existing wireless infrastructure. Despite its effectiveness, EM communication faces critical limitations in environments with high device density and security concerns. Li-Fi (Light Fidelity) presents a viable alternative by addressing interference, data security risks, and bandwidth constraints.

The widespread deployment of GHz-bandwidth communication in medical settings introduces significant security risks, as public Wi-Fi networks remain vulnerable to a growing number of sophisticated cyber threats [1]. Wi-Fi routers broadcast signals omnidirectionally, making unauthorized interception possible for any device within range of the transmission. Li-Fi mitigates this risk by confining data transmission to the visible light spectrum, which cannot penetrate opaque barriers, ensuring that data remains inaccessible outside the physical space of transmission[2].

Beyond security, Wi-Fi performance in hospital environments is further degraded by signal interference. In hospitals, thick concrete walls contribute to low-frequency signal reflection, and numerous electronic devices—such as laptops, anesthesia gas machines, medical imaging equipment, and life-support systems—compete within the same radio frequency spectrum [3]. In such high-density environments, where multiple devices must operate with minimal latency and maximum reliability, the interference inherent to Wi-Fi networks poses a serious limitation.

Li-Fi offers a more stable wireless alternative since visible light mitigates interference issues inherent to Wi-Fi. Unlike Wi-Fi, where most devices are calibrated to specific shared frequency bands for full network compatibility, Li-Fi can utilize the broader visible light spectrum, enabling transmission on distinct wavelengths with minimal interference. Baseline noise from ambient lighting can be mitigated through thresholding techniques that distinguish data signals from general illumination, while communication paths can be further isolated using opaque barriers to block unwanted signals or optimized with phase cancellation outside the transmission path to suppress interference. Additionally, since LED lighting primarily emits three distinct wavelengths—red, green, and blue—unused portions of the visible spectrum could be allocated for data transmission, reducing spectral overlap and enhancing signal clarity while preserving standard illumination[4]. In environments requiring strict control over signal integrity, integrating fiber optics with Li-Fi infrastructure could further enhance communication stability by providing high-speed, interference-free transmission channels. While some of these solutions remain theoretical in their application to Li-Fi, their feasibility is grounded in established optical and signal processing technologies, making Li-Fi a compelling alternative for secure, high-speed,

and interference-resistant wireless communication in environments where network reliability is critical.

For long-distance optical communication, Li-Fi is constrained by atmospheric effects such as spectral absorption and more so from group velocity dispersion. Spectral absorption, primarily due to atmospheric moisture, attenuates specific wavelengths, weakening signal intensity over distance. Group velocity dispersion, caused by variations in the refractive index of the atmosphere across different wavelengths, results in pulse broadening and signal distortion, ultimately limiting data rates and transmission range. Addressing these challenges requires high-power optical transmitters, adaptive optics, and advanced attenuation techniques to maintain signal integrity over extended distances [5]. While Li-Fi is not yet a universal replacement for Wi-Fi, its potential provides significant advantages where data security, network stability, and high-speed communication are critical. As research advances in mitigating dispersion and absorption effects, Li-Fi may become a more practical alternative to traditional EM-based communication.

Implementation: Lifi Communication

Our information transmission method employs character-to-binary 8-bit conversion (ASCII), the standard encoding system used by computers to interpret text. Our implementation transmits each character as an 8-bit binary sequence using a visible light source. The sender converts characters to their binary ASCII representation, transmitting a sequence of high and low light pulses corresponding to 1s and 0s. On the receiving end, a photoresistor detects variations in light intensity and reconstructs the binary sequence. Our photoresistor operated in a light-noisy environment, containing flashing red and blue light pulses. Despite this interference, the system maintained high accuracy due to its calibration for green light transmission. To achieve this, we developed a script that continuously sampled values from the analog input (A0), which measures photoresistor activity on a 0–1023 scale (0 = dark, 1023 = bright). By placing an LED screen in front of the sensor and displaying pure RGB primary colors (hex values ff0000, 00ff00, 0000ff), we determined that green light averaged 900, blue 700, and red 750, making green the optimal choice for communication with the hardware, and allowing us to set an optimized detection threshold of 750 for distinguishing transmitted signals from noise. A critical challenge in this system was synchronization, as timing discrepancies between transmission and reception could result in translation errors that completely alter the message. To address this, both codes execute a one-time synchronization sequence: the transmitter emits a 900ms sustained light pulse, and the receiver scans for an intensity above 750 to register the start of communication. Once detected, the receiver pauses for 2 seconds, while the transmitter waits 1.9 seconds before beginning transmission. This intentional timing discrepancy accounts for the persistence of vision artifacts and ensures that the receiver remains in phase with the transmitter. While high bit-rate transmission theoretically improves speed, reducing delay times by a factor of 10 resulted in bit blending, where alternating sequences like 10101010 were misinterpreted as 11111111 due to insufficient separation between pulses. This trade-off emphasizes the importance of balancing transmission speed with signal integrity to prevent data corruption.

Proposed Improvement: Redundancy Packet Encoding

To integrate this system into actual hardware, scalability presents a significant challenge, particularly in maintaining synchronization over extended transmissions. The receiver operates within a strict sampling window to capture signals, meaning any signal outside this window is effectively lost. This results in two primary translation errors: complete data loss and incorrect bit alignment, which can entirely change the interpreted character. To mitigate these issues, the receiver should ideally record periodically throughout all transmission phases rather than relying on a fixed, cycled sampling window. However, continuous recording introduces its own challenge—ensuring that the receiver starts reading at the correct position to accurately reconstruct the original data. Without a fixed reference point, decoding errors can propagate, causing misalignment between transmitted and received data.

To address this issue, noncoding redundancy packets provide structured reference markers that ensure consistent synchronization across extended transmissions. These markers are inserted at defined intervals, acting as absolute reference points for the receiver to realign its reading phase. The encoding process expands each 8-bit sequence by doubling every bit, ensuring clear transitions and preventing ambiguity from consecutive 0s or 1s. A predefined 1010 redundancy packet is inserted between augmented sequences, and an additional 1010 marks the transmission end. This ensures that decoding remains unambiguous, as the 1010 pattern cannot occur naturally in the encoded data. Unlike traditional synchronization strategies that rely on timing constraints, this method ensures the receiver can always recover its reference point, even in the presence of missed or delayed pulses. While this approach increases redundancy by a factor of six, it is a necessary tradeoff for maintaining synchronization integrity, particularly in high-speed applications where drift becomes more pronounced. Future improvements could involve refining the position markers to allow dynamic resynchronization mid-transmission, further improving robustness for large-scale implementations of Li-Fi communication.

On the receiving end, the system first detects and removes all 1010 sequences, systematically stripping away the noncoding packets. What remains is the augmented bit sequence, which is then restored to its original form by removing every second bit (2n), reconstructing the initial data exactly as it was transmitted. This method guarantees that symbol boundaries remain intact, eliminating the need for timing-based synchronization mechanisms, which are highly susceptible to drift. By ensuring that redundancy packets cannot be replicated within the coding data, this approach prevents false synchronization errors, making decoding a straightforward process of pattern matching and reduction rather than requiring complex real-time timing adjustments. See Figures 1 & 2 for a visual demonstration of the decoding process.

A traditional Forward Error Correction (FEC) approach would likely rely on methods such as Hamming codes or Reed-Solomon codes, which are designed primarily for bit-flip correction—that is, recovering from noise-induced changes in individual bits. However, these methods assume that the length of the transmitted data remains intact and do not effectively handle deletion and insertion errors, which are the primary concern in this system. More relevant

to this problem, convolutional codes, and turbo codes rely on probabilistic reconstruction techniques such as maximum-likelihood estimation, which infer missing bits based on statistical models. While these methods are effective for large datasets with continuous streams of information, they are less practical for small-scale transmissions like this, where explicit separation between coding symbols provides a more deterministic solution. That said, convolutional error correction could be a valuable addition for larger datasets, particularly if this system were to scale into high-speed Li-Fi communication with longer messages or real-time data streams. The use of systematic redundancy rather than parity checks eliminates the need for iterative decoding algorithms, making this a computationally simple and robust solution for ensuring unambiguous data transmission.

While this method increases transmission sizes by a factor of six, this level of redundancy is the minimum required to ensure that coding bits remain distinct and unambiguous. The goal of noncoding redundancy packets is to introduce a separation between coding bits that cannot be mistaken for one another, eliminating the need for pulse duration-based synchronization, which is prone to drift. Future improvements could focus on introducing a noncoding start sequence, which would instruct the receiver when to start recording rather than relying on it to be active when the message begins. This would be particularly beneficial for higher refresh rate systems, where synchronization drift is more pronounced. However, this does not entirely eliminate synchronization concerns, as successful reception is still dependent on the relative phase of each program's refresh cycle. In theory, matching the clock speeds of the transmitter and receiver would mitigate synchronization issues further, but this has not been tested in the current system. Ultimately, this redundancy-based approach offers a deterministic and computationally simple solution to deletion and insertion errors, ensuring self-delimiting symbol transmission without requiring complex reconstruction algorithms. While bandwidth efficiency and transmission speeds are sacrificed, the tradeoff is justified for unambiguous, reliable, and easy-to-decode Li-Fi transmission.

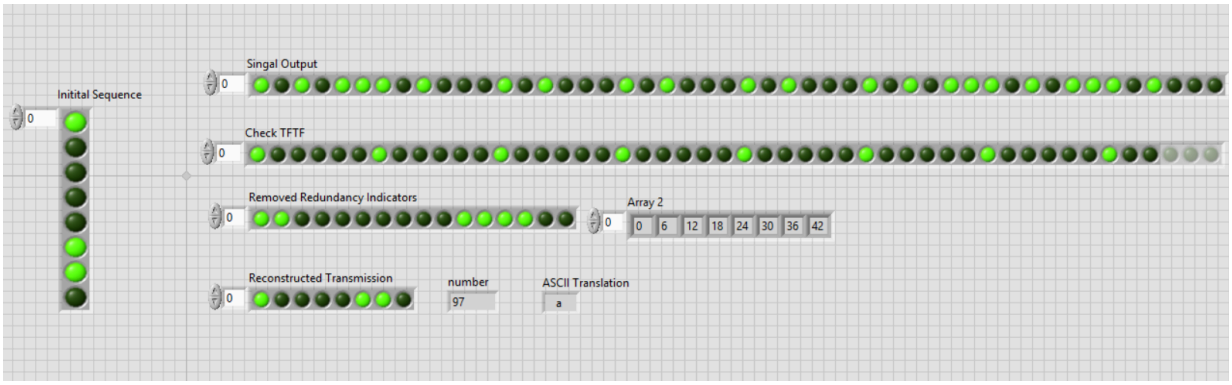


Figure 1: The **Initial Sequence** (left panel) shows the binary input **01100001** before encoding, representing the **original 8-bit ASCII character**. The **Signal Output** (top row) displays the received signal after **ADC**, where bit-doubling and noncoding redundancy packets are visible. Below, the **Check TFTF** step detects occurrences of the **1010 redundancy packet**, marking their start indices in **Array 2** for later removal. The **Removed Redundancy Indicators** row presents the **augmented message** after extracting noncoding packets, revealing the **doubled-bit version of the original transmission**. Finally, the **Reconstructed Transmission** restores the original **8-bit sequence** by removing every second bit (2n), allowing for conversion into a **decimal value (97) and its corresponding ASCII character ('a')**.

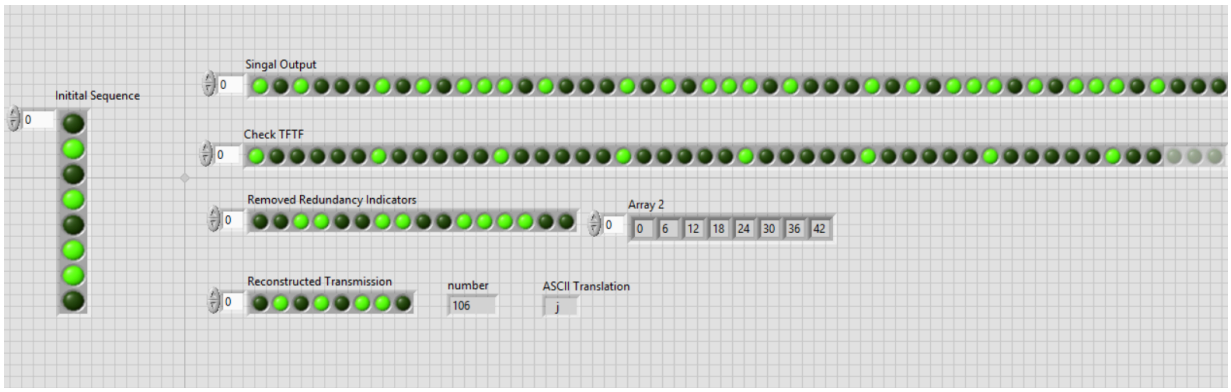


Figure 2: This figure follows the same decoding process as Figure 1 but for a different input character. The **Initial Sequence** (left panel) represents the binary input **01101010**, corresponding to ASCII **106 ('j')**. The **Signal Output** again contains **redundancy markers and bit-doubling**, which are detected in **Check TFTF** and logged in **Array 2**. The **Removed Redundancy Indicators** row strips these markers, revealing the **doubled-bit transmission**, which is then **reduced back to its original 8-bit form** in **Reconstructed Transmission**, confirming the final output as **'j'**. These figures demonstrate the **effectiveness of redundancy packets in preventing synchronization errors**, ensuring **clear character reconstruction** from Li-Fi signal data.

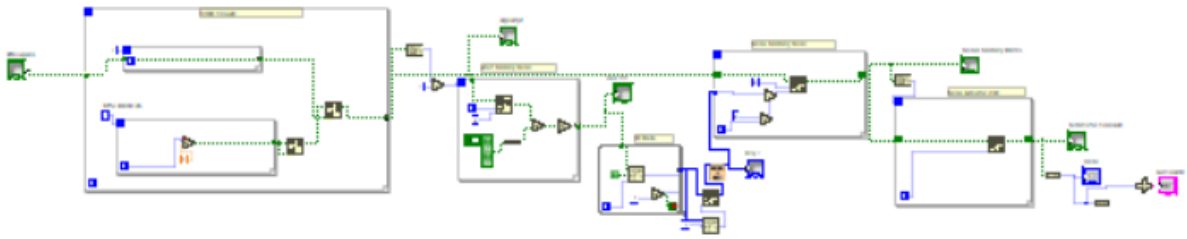


Figure 3: The full block diagram of the **encoder** and decoder **algorithms**, illustrating the complete signal processing pipeline used for transmitting and reconstructing binary-encoded Li-Fi messages. The leftmost section handles initial **encoding**, including **bit-doubling** and **redundancy packet insertion**. The central section processes the received signal, **detecting** noncoding redundancy markers and extracting valid data bits. The final section on the right performs **binary-to-decimal conversion**, reconstructing the original ASCII character from the received signal.

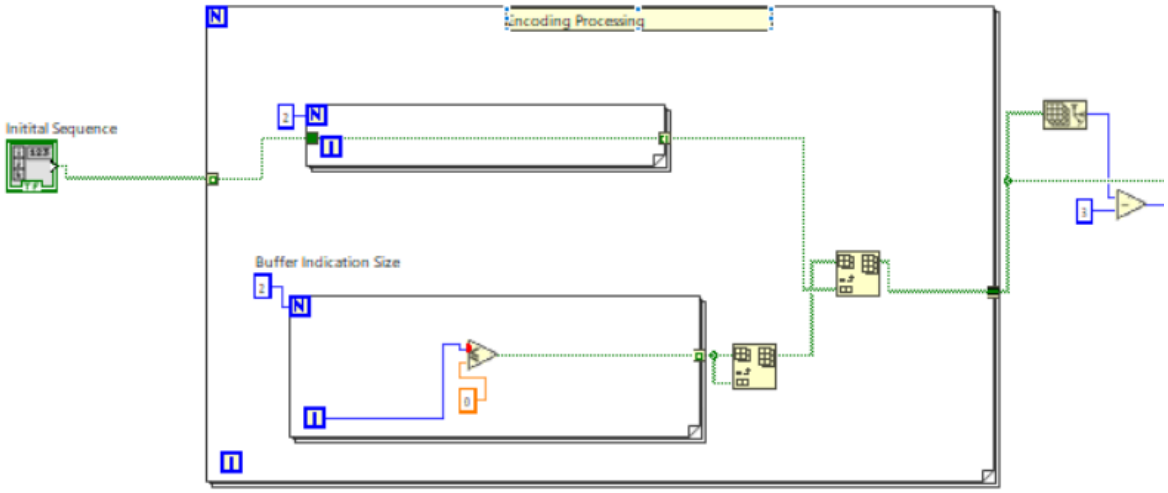


Figure 4: Block diagram of the **encoder**. The Initial Sequence input is processed through bit-doubling and redundancy packet insertion. The Buffer Indication Size section manages the insertion of predefined 1010 sequences.

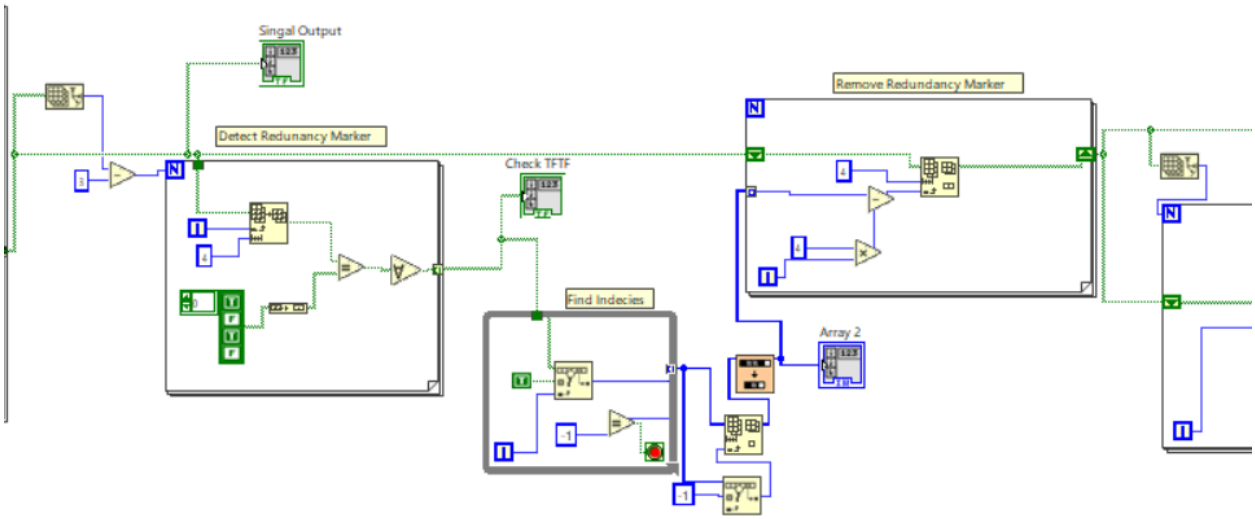


Figure 5: Block diagram of **redundancy marker detection and deletion**. The Check TFTF module scans for predefined 1010 redundancy markers, while the Find Indices section records their positions in Array 2. These markers are then systematically **removed** in the Remove Redundancy Marker module, leaving only the augmented data sequence. The Remove Augmented String step **extracts** the original binary message by eliminating duplicate bits, reconstructing the intended 8-bit transmission. Finally, the processed data is converted into its decimal and ASCII representations, ensuring accurate recovery of the transmitted character.

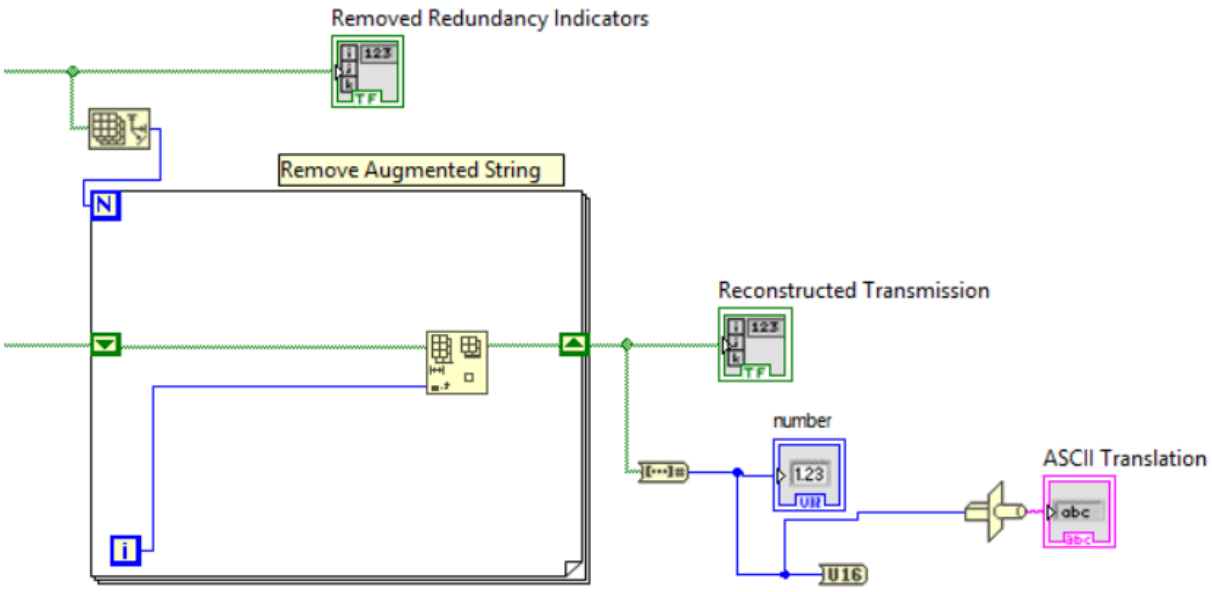
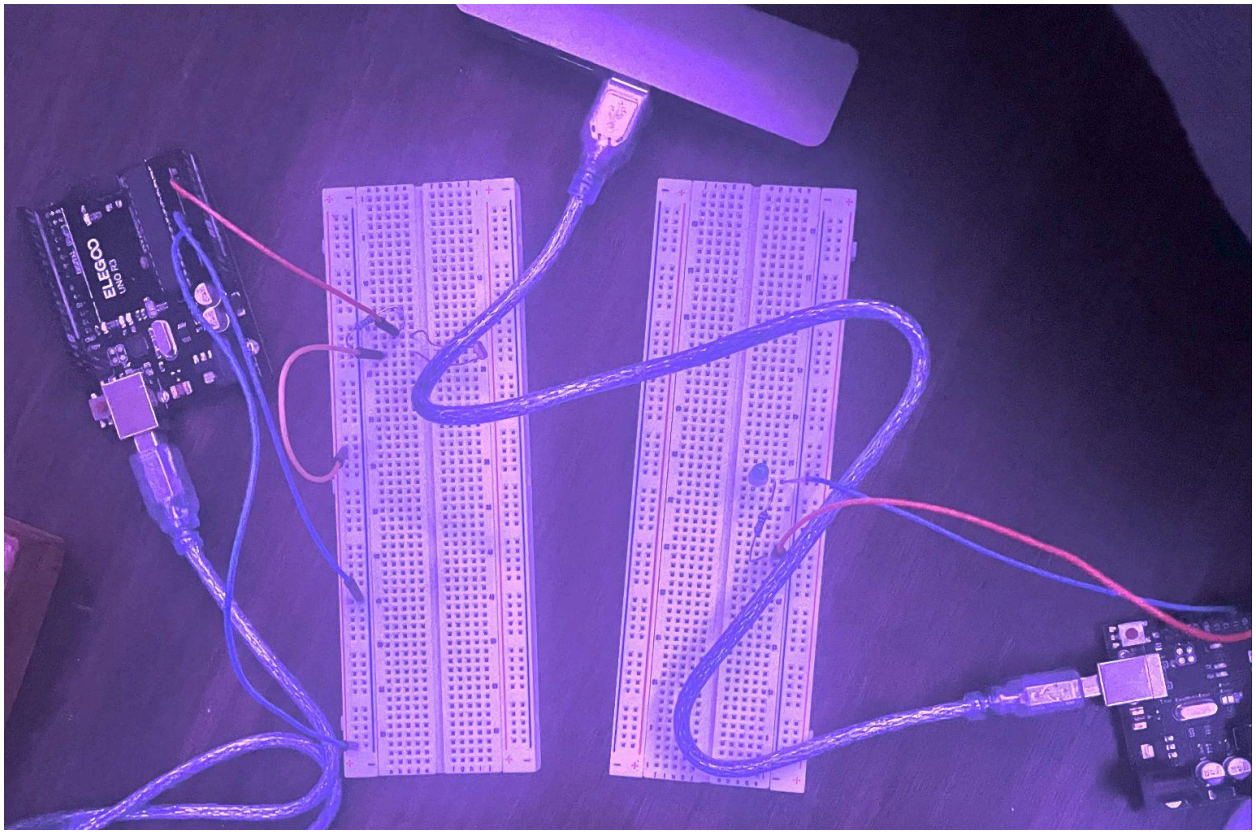


Figure 6: Block diagram of reverse augmenting the signal. After redundancy markers have been removed, the Remove Augmented String module **reomves** the doubled bits, restoring the data to its original form. The Reconstructed Transmission output is then **converted** from binary to decimal, followed by an ASCII translation step that maps the decimal value to its corresponding character representation.



References:

[1] Aura, "The Dangers of Public Wi-Fi: How Hackers Can Steal Your Data & How to Stay Safe," *Aura.com*, [Online]. Available: https://www.aura.com/learn/dangers-of-public-wi-fi.

[2]G. Blinowski, "Security issues in visible light communication systems," in IFAC, Warszawa, Poland, 2015
https://www.academia.edu/109808083/Security_issues_in_visible_light_communication_systems

[3] 7signal, "WLAN Woes: Why Wi-Fi Fails in Hospitals," *7signal.com*, [Online]. Available: http://7signal.com/news/blog/wlan-woes-why-wi-fi-fails-in-hospitals#:~:text=Challenging%20physical%20environment.,to%20its%20standard%20computer%20system.

[4] Alfattani, Safwan. "Review of LiFi Technology and Its Future Applications "Journal of Optical Communications, vol. 42, no. 1, 2021, pp. 121-132. https://doi.org/10.1515/joc-2018-0025

[5] A. Singh and P. Kumar, "Atmospheric Propagation Limitations for Free-Space Optical Communications," *arXiv*, Apr. 2021. [Online]. Available: https://arxiv.org/pdf/2104.00611.

Send code:

```
// Transmitter Code: Sends binary data using LED



const int ledPin = 9;  // LED connected to pin 9

String message = "11100010";  // Example binary message to send

int delayTime = 1000;  // Delay between bits (milliseconds)

int BUiginSequence=1;



void setup() {

  pinMode(ledPin, OUTPUT);



}



Void loop() {  //the great beginning of the codeatholon that it took to
make this simple thing haha.
```

```
if (BUiginSequence==1) { //this if loop makes it so the inisilization only runs once.
  delay(10000);
  digitalWrite(ledPin,HIGH);
delay(900);                          //These delays are in time with the other computer to allow syncing between them
digitalWrite(ledPin,LOW);
delay(1000);
BUiginSequence=0; //sets the value to 0 to break the loop and not run the inisilization proccess again
}




  for (int i = 0; i < message.length(); i++) {
    if (message[i] == '1') {
      digitalWrite(ledPin, HIGH);  // Turn LED ON
    } else {
      digitalWrite(ledPin, LOW);  // Turn LED OFF
    }
    delay(delayTime);  // Wait for the next bit
  }
  delay(5000);  // Delay before repeating the message
```

```
}
```

Receive code:

```
// Receiver Code: Reads light pulses and decodes binary data


const int sensorPin = A0;  // Light sensor connected to analog pin A0

const int threshold = 750;  // Threshold to detect light intensity

int startRead = 0;

String receivedMessage = "";


void setup() {

  Serial.begin(9600);  // Start serial communication

  pinMode(sensorPin, INPUT);


}
```

```
void loop() {

  int sensorValue = analogRead(sensorPin);  // Read light sensor value



      if (sensorValue > threshold && startRead == 0) { // when the sensor
recieves the inisilization light, syncing begins.

      Serial.println("reading");                        // just a bit to
see that the computer is actually reading

      delay(2000);                                      //delay allows for
computers to be intime with eachother.

      startRead = 1;

     }

  if (startRead == 1) {

    sensorValue = analogRead(sensorPin);  // Read light sensor value



    if (sensorValue > threshold) {

        receivedMessage += "1";  // Light detected, make a 1



    } else {

        receivedMessage += "0";  // No light detected, makes a 0

    }

      Serial.println(receivedMessage);



    delay(1000);  // Wait for the next bit

    if (receivedMessage.length() >= 8) {  // Ensure full 8-bit message,
nothing less.
```

```
        int decimalValue = binaryToDecimal(receivedMessage);  // Convert
to decimal

        char asciiChar = (char)decimalValue;  // Convert decimal to ASCII



        Serial.print("Received Binary: ");

        Serial.print(receivedMessage);

        Serial.print(" | Decimal: ");

        Serial.print(decimalValue);

        Serial.print(" | ASCII: ");

        Serial.write(decimalValue);  // Print ASCII character directly

        Serial.println();  // New line for clarity



        receivedMessage = "";  // Clear message for the next cycle

        delay(5000);  // Wait before restarting

    }

}

}

// Function to convert an 8-bit binary string to decimal

int binaryToDecimal(String binary) {

    int decimalValue = 0;



    for (int i = 0; i < 8; i++) {

        if (binary[i] == '1') {
```

```
            decimalValue |= (1 << (7 - i));  // Bitwise conversion from
string to decimal.

        }

    }



    return decimalValue;

}
```

10101010 Test

11100010 Test

\